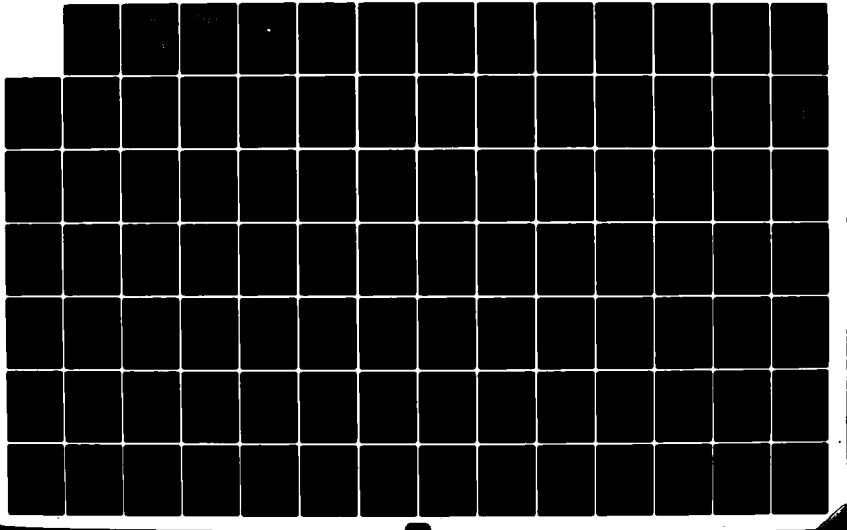
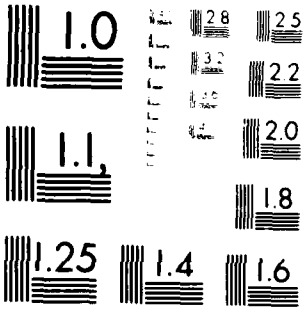
AD-A131 941 A SOFTWARE ENGINEERING ENVIRONMENT FOR THE NAVY(U) 1/3
NAVAL MATERIAL COMMAND WASHINGTON DC 31 MAR 82

UNCLASSIFIED F/G 9/2 NL





A SOFTWARE ENGINEERING ENVIRONMENT FOR THE NAVY

REPORT OF THE NAVMAT SOFTWARE ENGINEERING ENVIRONMENT WORKING GROUP

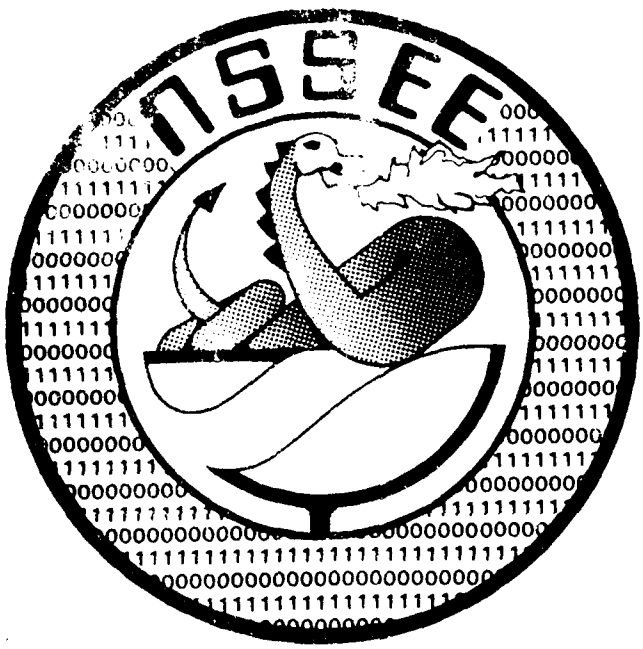
March 31, 1982

A SOFTWARE ENGINEERING ENVIRONMENT FOR THE NAVY

REPORT OF THE NAVMAT
SOFTWARE ENGINEERING ENVIRONMENT
WORKING GROUP

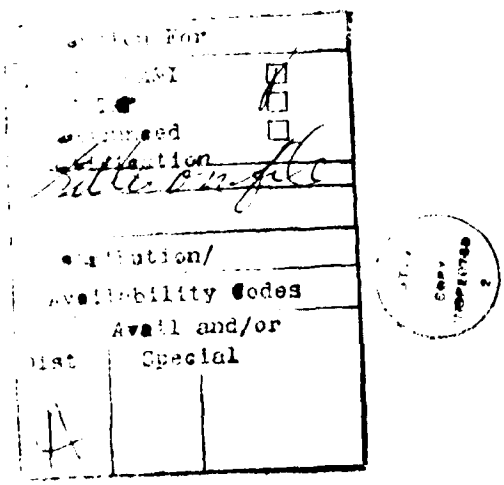
March 31, 1982

EXECUTIVE SUMMARY



SOFTWARE ENGINEERING ENVIRONMENT

WORKING GROUP



March 31, 1982

The logo for the Navy Standard Software Engineering Environment (NSSEE) shows a sea serpent riding the crest of the software technology wave and breathing structure into the software development process (represented by the well-ordered field of ones and zeros). NSSEE should be pronounced "Nessie" in honor of a lake-dwelling resident of the British Isles. History does not record any interaction between Nessie and another well-known British citizen, Ada. The modern Ada and NSSEE will be well-acquainted, however.

PREFACE

This is one of a set of documents resulting from the work of the Naval Material Command's Software Engineering Environment Working Group (SEEWG). The set includes an "Executive Summary," "Framework for a Navy Standard Software Engineering Environment," and "Evolution Plan for a Navy Standard Software Engineering Environment."

The Framework document provides a basis on which to plan and develop a standard methodology-driven software engineering environment for the Navy. The Evolution Plan describes a strategy for transition to a standard software engineering environment and also the evolution of the environment itself.

These documents represent the culmination of the first phase of an effort to develop standard tools and procedures to support development of software for Navy embedded computer systems. It is intended ultimately to implement a standard software engineering environment encompassing the whole software life cycle and supporting effective use of the Ada programming language. In the interim, these documents will provide a basis for co-ordinating decisions with respect to improvements to existing software support systems.

FOREWORD

This document contains information on a framework for a software engineering environment. It is the work of the NAVMAT Software Engineering Environment Working Group (SEEWG). The SEEWG studied the issues of what is the life cycle of software, what is a software engineering process, what is a software engineering environment, and what methodologies, tools and techniques are applicable.

The Framework is based primarily on "Software Engineering Environments for Navy Embedded Computer Systems," February 27, 1982, prepared for NAVSEA 61R by Software Architecture and Engineering, Inc., Arlington, Virginia.

Other documents used for source information include:

1. Top Level Requirements for Navy Minimum Ada Programming Support Environment, 4 December 1981.

2. The Software Functional Description for the Software Production Facility of the NAVELEX C^3 Software Support Facility, 29 September 1981, Software and Computer Directorate, Naval Air Development Center, Warminster, Pennsylvania.

EXECUTIVE SUMMARY

The Software Engineering Environment Working Group (SEEWG) was established by the Chief of Naval Material (MAT 08Y) on April 27, 1981. Its purpose was to coordinate current and planned software engineering environment (SEE) initiatives in the naval systems commands to ensure that they are well integrated and non-redundant. Further, it was to provide a NAVMAT focal point for addressing software engineering environment issues and to coordinate action on critical SEE related efforts.

The specific objectives defined in the SEEWG charter were:

A. Provide a focal point for SEE activities within the Naval Material Command (NAVMAT).

B. Coordinate current and planned SEE projects within NAVMAT to ensure consistency, compatibility, and non-redundancy of effort.

C. Develop a specification for Navy SEEs that is based on the Navy Minimum Ada Programming Support Environment (MAPSE).

D. Guide the assimilation and integration of standard SEEs into the Navy software development process.

The working group formed to address these objectives was composed of designated managers and technical experts from naval systems commands and field activities (listed in Appendix A). In addition, the SEEWG obtained contracted support from Software Architecture and Engineering, Inc. (formerly Lesko/Fox Associates) of Arlington, Virginia. A series of meetings was held to address the technical, programmatic, and policy issues.

The objective of providing a focus for Navy software engineering initiatives has been achieved.

The SEEWG members themselves represent a number of the principal software engineering projects within the Navy. Their common effort to define the nature of a standard software engineering environment for the Navy

has received significant recognition in both the Navy software development world and in industry. Momentum towards a new software engineering process for the Navy has been created which can be exploited to assure the development and introduction of new tools and techniques which have been identified as needed. NAVMAT and a re-chartered SEEWG will manage this process to capitalize on the progress thus far.

The objectives of developing requirements for a MAPSE-based software engineering environment is addressed in the SEEWG report "Framework for a Navy Standard Software Enginering Environment." It should be noted that the Framework proposed is preliminary to formal specification of requirements for acquisition purposes.

The objectives of guiding assimilation and integration of standard SEE's into the Navy software development process has been addressed both in the Framework document and in the "Evolution Plan for a Navy Standard Software Engineering Environment." This SEEWG product deals specifically with the transition process from the current situation to the future standard SEE and also with subsequent technological innovation.

Thus, the two tangible products of the SEEWG effort are the Framework and Evolution Plan documents. The main conclusions of each are summarized as follows.

Framework for a Navy Standard Software Engineering Environment:

- The software engineering environment must support the entire software life cycle. A life cycle model has been developed in which to view software development as an incremental process.

- The software engineering environment must be methodology-driven. A collection of tools is of little benefit in the absence of an integrating discipline for their employment.

- The tools and techniques to support each activity in the software life cycle have been identified.

- The information products derivable in each activity of the life cycle have been categorized according to whether the information must be baselined in the software engineering environment data base or not. Baselined products are configuration managed and persist over the life of the software system being developed. Thus, the nature of the environment's data base is fundamental to the choice of tools provided and to the integrating discipline for their employment.

Evolution Plan for a Navy Standard Software Engineering Environment:

- The Navy Standard Software Engineering Environment (NSSEE) will be based on the Minimum Ada Programming Support Environment (MAPSE). The MAPSE provides the best foundation upon which to build a fully-integrated software engineering environment.

- Existing Navy application projects and support systems cannot be ignored and should be able to gain some benefit from the concepts and products resulting from the definition and development of the NSSEE. An approach is described to assure the smooth transition to a modern Navy Standard Software Engineering Environment.

- The Navy must build at least one integrated tool set that will operate through all phases of the life cycle model. The availability of such a Navy Standard Software Engineering Environment and the control afforded by the baselining of standardized information products will greatly improve the Navy's ability to manage the acquisition and maintenance of effective software systems at reduced life cycle costs.

- Current policies, standards, and guidelines need to be changed in order to provide a framework within which a NSSEE can be built and used. Evolution of these policies, standards, and guidelines will be necessary as the Navy Standard Software Engineering Environment evolves.

- Training/education is extremely important for managers, developers, and users of the NSSEE. Understanding how to use the Navy Standard Software Engineering Environment competently is crucial to deriving any benefits from it.

- The need for a research and development effort is identified to ensure the timely maturing of software technology to support full implementation of the desired Navy Standard Software Engineering Environment.

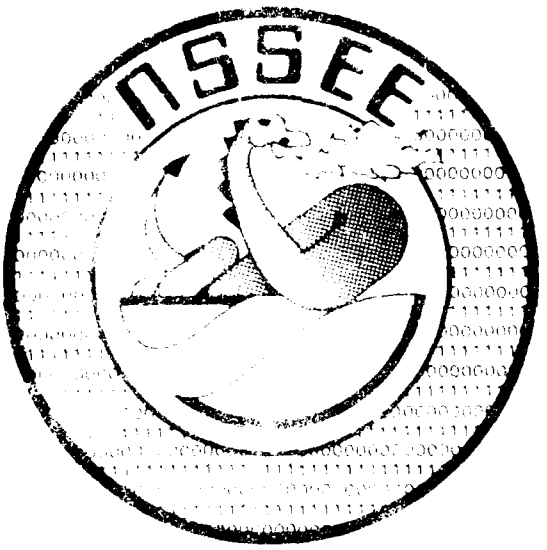
Perhaps the most important result of the SEEWG effort, however, may be the attention it has drawn to the need for a standard software engineering process to support the effective use of the Ada programming language in Navy applications. Momentum for a modern Navy Standard Software Engineering Environment (NSSEE) has been gathered. Action by NAVMAT will now be taken to see that the SEEWG recommendations give rise to actual specifications for a NSSEE and to policies and standards appropriate to support it.

APPENDIX A

SEEWG MEMBERSHIP

LCDR K. Paige (Co-chairman)	PMS 408
CDR R. Simpson (Advisor)	OP-942
CDR A. Hadley (Advisor)	OP-942
LCDR J. Kramer	MAT 08Y/AJPO
R. Eyres	NOSC
H. Stuebing	NADC
CDR R. Goodman	PMS 408
J. Blackmon	PMA 533
T. Conrad	NUSC
LCDR J. Barnes	NAVELEX (ELEX-814)
D. Jordan	PME 120-3
LTCOL A. Hesser	Marine Corps
T. Phillips	NOSC
S. Peele	FCDSSA Dam Neck
P. Andrews	NAVSEA (SEA 61R)
J. Machado	NAVELEX (ELEX 6134)
B. Zempolich	NAVAIR (AIR 360B)
T. Dawson (Recorder)	NUSC
R. Charette	NUSC
C. Russ	FCDSSA Dam Neck
[illegible] Balmer	FCDSSA San Diego
[illegible] Linley	NAC
[illegible] Potter	NAVELEX (ELEX-814)
W. [illegible]	NSWC, Dahlgren
[illegible] (Consultant)	Software Architecture & Engineering, Inc.

FRAMEWORK FOR A NAVY STANDARD SOFTWARE ENGINEERING ENVIRONMENT



SOFTWARE ENGINEERING ENVIRONMENT

WORKING GROUP

March 31, 1982

The logo for the Navy Standard Software Engineering Environment (NSSEE) shows a sea serpent riding the crest of the software technology wave and breathing structure into the software development process (represented by the well-ordered field of ones and zeros). NSSEE should be pronounced "Nessie" in honor of a lake-dwelling resident of the British Isles. History does not record any interaction between Nessie and another well-known British citizen, Ada. The modern Ada and NSSEE will be well-acquainted, however.

PREFACE

This is one of a set of documents resulting from the work of the Naval Material Command's Software Engineering Environment Working Group (SEEWG). The set includes an "Executive Summary," "Framework for a Navy Standard Software Engineering Environment," and "Evolution Plan for a Navy Standard Software Engineering Environment."

The Framework document provides a basis on which to plan and develop a standard methodology-driven software engineering environment for the Navy. The Evolution Plan describes a strategy for transition to a standard software engineering environment and also the evolution of the environment itself.

These documents represent the culmination of the first phase of an effort to develop standard tools and procedures to support development of software for Navy embedded computer systems. It is intended ultimately to implement a standard software engineering environment encompassing the whole software life cycle and supporting effective use of the Ada programming language. In the interim, these documents will provide a basis for co-ordinating decisions with respect to improvements to existing software support systems.

FOREWORD

This document contains the executive summary of the work done by the NAVMAT Software Engineering Environment Working Group (SEEWG). For the past year, the members of the SEEWG have been grappling with the issues of what a software engineering environment is, what an environment contains, how to acquire one that meets the Navy's needs, and how to deal effectively with the transition to the new software engineering technology.

Table of Contents

Introduction

1.1 Purpose of the Framework.....1
1.2 Background.....2
1.3 Guiding Principles.....3
1.4 Software Engineering Interface Standards....4
1.5 Structure of the Framework.....6

Part I: Context of Software Engineering

1. Life Cycle Model.....I-2
1.1 A Current Life Cycle Model.....I-4
1.2 A Recommended Life Cycle Model.....I-7
1.2.1 Incremental Development.....I-7
1.2.2 Early Prototyping.....I-9
1.2.3 Extended Correctness Analysis.....I-12
1.2.4 Management Integration.....I-12
1.3 Relation to Acquisition Management Models....I-12
1.4 SEE Requirements Derived from the Life Cycle Model.....I-15

2. Methodologies and Tools.....I-16
2.1 Requirements Analysis.....I-18
2.1.1 Characteristics.....I-19
2.1.2 Methodologies.....I-21
2.1.3 Supporting Tools.....I-23
2.1.4 Requirements on the SEE.....I-24

2.2 Specification.....I-25
2.2.1 Characteristics.....I-25
2.2.2 Methodologies.....I-26
2.2.3 Supporting Tools.....I-28
2.2.4 Requirements on the SEE.....I-29

2.3 Design.....I-29
2.3.1 Characteristics.....I-30
2.3.2 Methodologies.....I-31
2.3.3 Supporting Tools.....I-33
2.3.4 Requirements on the SEE.....I-33

2.4 Implementation.....I-33
2.4.1 Characteristics.....I-34
2.4.2 Methodologies.....I-35
2.4.3 Supporting Tools.....I-38
2.4.4 Requirements on the SEE.....I-38

2.5 Correctness Analysis.....I-41
2.5.1 Characteristics.....I-41

2.5.2 Methodologies.....I-44
2.5.3 Supporting Tools.....I-46
2.5.4 Requirements on the SEE.....I-49

2.6 Management.....I-50
2.6.1 Characteristics.....I-50
2.6.2 Methodologies.....I-53
2.6.3 Supporting Tools.....I-54
2.6.4 Requirements on the SEE.....I-54

3. An Abbreviated Software Engineering Process.....I-55

Part II: Description of A Software Engineering Environment

1. Data Base.....II-3
1.1 Contents.....II-4
1.1.1 Baselined Products.....II-5
1.1.2 Non-Baselined Data.....II-11
1.1.3 Measurement Data.....II-16
1.1.4 Archive Data.....II-19
1.1.5 Support System Library.....II-20

1.2 Relations Among Data Base Objects.....II-21
1.2.1 Macro Relations.....II-22
1.2.2 Micro Relations.....II-23

2. Support System.....II-27
2.1 External Interfaces.....II-28
2.1.1 Command Language.....II-28
2.1.1.1 Characteristics.....II-29
2.1.1.2 Capabilities.....II-32
2.1.2 Characteristics of SEE Hardware.....II-34
2.1.3 Target System Interfaces.....II-37

2.2 Functional Capability.....II-37
2.2.1 System Functions.....II-38
2.2.1.1 Multi-User Interactive Support.....II-38
2.2.1.2 Background Processing.....II-39
2.2.1.3 Data Base Management.....II-39
2.2.1.4 Pipelining and Redirection..II-40
2.2.1.5 Operational Control.....II-40
2.2.1.6 Remote Access.....II-41
2.2.1.7 Access and Resource Control.II-42
2.2.1.8 Utilities.....II-42

2.2.2 Software Engineering Functions.....II-43
2.2.2.1 General Services.....II-43
2.2.2.2 Activity-Specific Tools.....II-48
2.2.2.3 Application Paradigms.....II-57

2.3 Performance Requirements.....II-58

2.4 Design Constraints.....II-60
2.4.1 Independence.....II-60
2.4.2 Architectural Flexibility.....II-61
2.4.3 Ada Compatibility.....II-61
2.4.4 Generalized Tool Interface.....II-61

Appendix A: Some Background on Software Development

1. State-of-the-Practice in the Navy.....A-1
1.1 A Comparison of Two Systems.....A-2
1.2 Currently Available Support Systems.....A-4
1.3 A Case Study.....A-5

2. Modernization and R&D Activities.....A-6
2.1 Modernization of Support Systems.....A-10
2.2 Software Engineering R&D.....A-12

List of Figures

Figure No.	Title	Page No.
1.3-1	Life Cycle Activity Interfaces	[illegible]
1-1	The Life Cycle of Software	I-3
1-2	A Current Life Cycle Model	I-5
1.2.1-1	Incremental Development Model	I-10
1.2.1-2	An Incremental Sub-Model	I-11
1.3-1	Joint Logistics Commanders Proposed Life Cycle Acquisition Model	I-14
2-1	The Starting Point of the Software Engineering Process within a Weapon System Life Cycle	I-17
2.3.3-1	Methodologies and Tools of Design	I-34
2.5-1	The Scope of Correctness Analysis	I-42
2.5.1-1	Span of Correctness Analysis	I-43
2.5.3-1	Methodologies and Tools of Correctness Analysis	I-47
2.6-1	The Management Process	I-51
1.2.2-1	Examples of Micro Relations among Baselined Products	II-26
2.1.2-1	SEE User's Classes	II-36
2.2.2.2-1	Flow of Requirements Analysis: Activity to Methodology to Tool	II-50
2.2.2.2-2	Flow of Specification: Activity to Methodology to Tool	II-51
2.2.2.2-3	Flow of Design: Activity to Methodology to Tool	II-52

2.2.2.2-4	Flow of Implementation: Activity to Methodology to Tool	II-53
2.2.2.2-5	Flow of Correctness Analysis: Activity to Methodology to Tool	II-54
2.2.2.2-6	Flow of Management: Activity to Methodology to Tool	II-55
2.2.2.2-7	Pipelining for Activity-Specific Tools	II-56
A.1.1-1	A Comparison of Two Operational Systems	A-3
A.1.3-1	Application of Automated Support	A-7
A.1.3-2	Workload vs. Labor Resources	A-8
A.1.3-3	Workload vs. ADP Equipment Resources	A-9

INTRODUCTION

INTRODUCTION

The Software Engineering Environment Working Group (SEEWG) was created by the Naval Material Command (NAVMAT) in April of 1981. Its purpose was to develop a strategy for software engineering environments in the Navy. The goal of the SEEWG was to eliminate the duplication of effort in support systems for software development and to improve software development productivity and the reliability of software. Specifically, its objectives were:

1. To provide a focal point for SEE activities within NAVMAT.

2. To coordinate current and planned SEE projects to ensure consistency, compatibility and non-redundancy of effort.

3. To guide the assimilation and integration of standard SEEs into the Navy software development process.

4. To develop a specification for Navy SEEs that is based on the Navy Minimum Ada Programming Support Environment (MAPSE).

1.1 Purpose of the Framework

The Framework section presents the standard SEE in terms of the methodologies and tools required to support the Navy's software life cycle management. It is preliminary to developing a formal specification of SEE requirements for acquisition. It will provide some ground rules and a focus for the Navy's SEE efforts, both long and short term.

Part I of the Framework outlines a well-defined doctrine for the software engineering discipline. This provides a context for the Navy SEE framework.

Part II defines the characteristics of a SEE. Some background in the state-of-the-art practice of software in the Navy is provided in Appendix A of the Framework. The background information suggests the rationale for a SEE program.

The Navy will benefit from starting a program to implement, in a standard manner across the Navy, some of the proven software engineering methodologies and tools. Their introduction will begin to prepare Navy personnel for the Navy standard SEE. It will help to improve software reliability and the productivity of software development. It will also begin standardization on product forms at the activity interface points described in Section 1.4.

More research, analysis and development is needed to improve the less well understood methodologies, techniques and tools of software engineering. The Framework identifies a number of these items so that additional work can begin to meet Navy requirements. These can be introduced to Navy personnel as they are refined and become available for use. Emerging technologies such as knowledge-based expert systems are expected to find many applications but the technology is still undeveloped.

This document is intended for a wide audience in the Navy. The NAVMAT planning and policy hierarchy should view it as a target for software engineering planning and standards redefinition. Support software production groups should focus on the software engineering process, methodologies and tools outlined to guide their short-term decisions on support system upgrades. Acquisition managers should be using the document to guide procurement decisions relating to software engineering and software engineering environments.

1.2 Background

The current state of software engineering practice in the Navy exhibits a number of serious weaknesses; the basic concepts reflected in the Framework are intended to counter these practices. Three points summarize the situation discussed more fully in Appendix A:

> Variation in Practice - There is a wide range of software engineering practices within the Navy and among its contractors. These practices lag the state-of-the-art by as many as fifteen years. The same variation is evident in the support systems used.

As a result, software is difficult to manage, software products vary widely in cost, reliability and maintainability, and it is extremely difficult to introduce advances in technology on a Navy-wide basis to improve either productivity or the overall quality of the software.

Need to Increase Productivity - Based on the analysis of projected work loads, the demand for software will soon outstrip the resources available for producing software. This is a potential threat to fleet readiness. The only viable answer is to increase productivity, which can only be accomplished on the scale required by increasing the productivity of the process as a whole.

Need for Focus - One way to increase productivity is to improve the methodologies and tools used by the production groups. The Navy has a large number of, as yet, uncoordinated efforts to modernize its support systems. These efforts must be better focused to make a significant impact on both productivity and quality.

1.3 Guiding Principles

The Framework provides a means to improve the current situation. The Framework was created with several fundamental principles driving the work. These principles should be kept in mind when reading this document. They are:

Based on Navy MAPSE - The Navy SEE must be consistent with the Navy MAPSE. The SEE must include full support for Ada plus an integrated data base. It must also provide continuing support for existing Navy standard programming languages.

Methodology-Based - The software engineering process and its related methodologies are the foundation for the SEE and its methodologies and tools. The entire life cycle of software is included in the software engineering process. Each methodology is related to specific activities of the process. The methodologies and tools are divided into those that support crea-

tivity and those that support the recording of products. (The importance of splitting creativity and recording is discussed in Part I.) The concept of having formal interfaces defined for each software engineering activity is fundamental to standardization as discussed in Section 1.4.

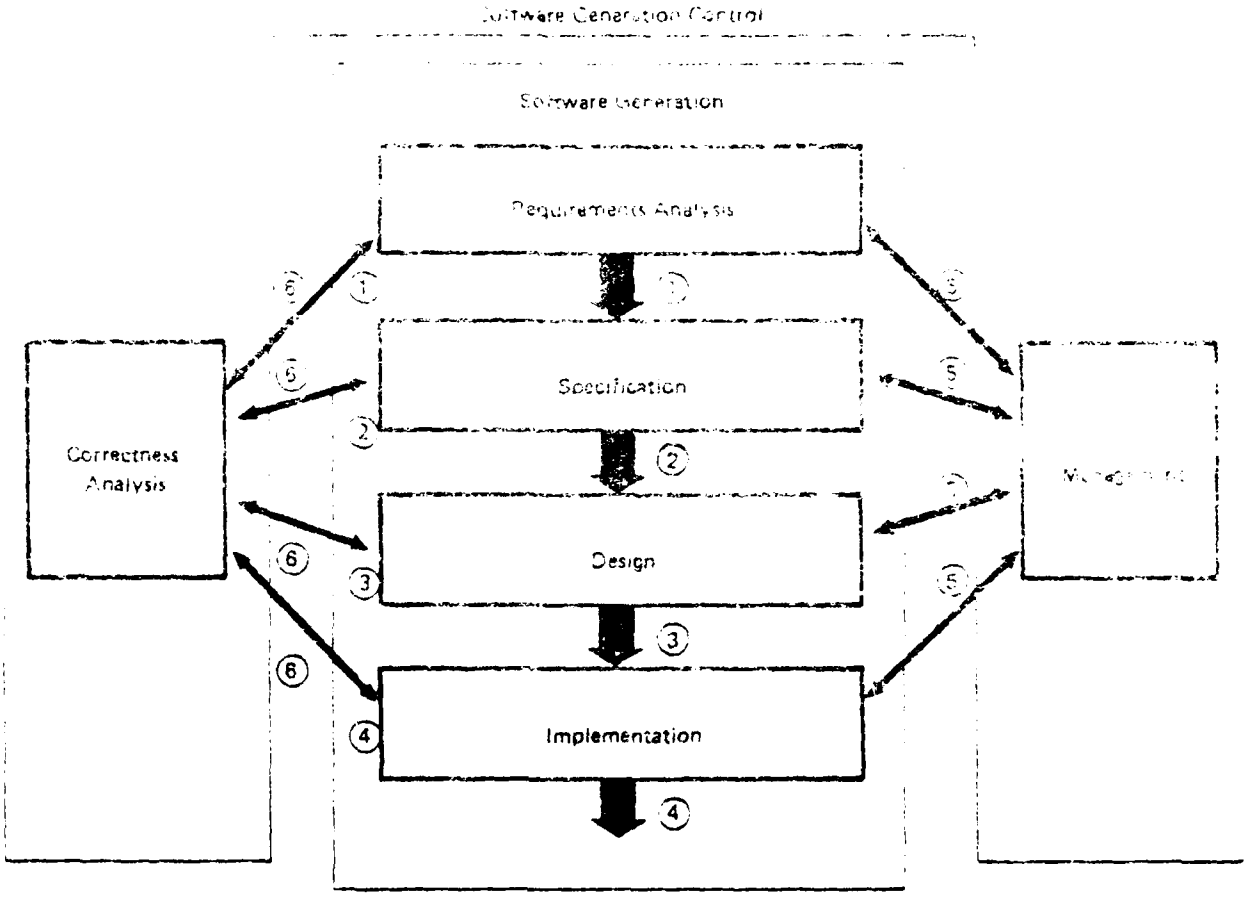
[illegible]

resources to support the SEF and its projected workload effectively and efficiently.

1.4 Software Environment Interface Standards

One of the first priorities of the Navy SEE program, as identified by the SEEWG, is the standardization of the form of software and its supporting information products. Software should be transportable at minimum cost between one support group (and environment) to another support group (and possibly a different environment). This standardization will be achieved by defining the formal interfaces between software life cycle activities, as defined in Part I Section 1.2.

The concept of formal interfaces is shown in Figure 1.4-1. Software production and life cycle support can be



INTERFACE BASELINED PRODUCTS KEY:

1. Semantic information
2. Specification
3. Design
4. Source text, object modules, SYSGEN data, etc.
5. Development plan, ECR/TR log, etc.
6. Test plan, test procedures, review reports, etc.

Figure 1.4-1: Life Cycle Activity Interface

divided into two activities: software generation and the control of software generation. Software generation consists of the four activities defined in Part I: requirements analysis, specification, design and implementation. "Interfaces" between activities are the baselined products created for the activity. These "interface" products are used by others to work toward the creation of the final software product.

The two activities of software generation control are correctness analysis and management. The "interfaces" between the control activities and the software generation activities are a variety of plans and measurements, most of which are baselined products.

The baselined products that are the formal interfaces are defined in Part II Section 1.1.1. Several of these products are identified in Figure 1.3-1. Specifications for these formal interfaces will be part of the SEE program. These specifications will define the form, content and physical configuration of interfaces, which will simplify moving them from one environment to another.

The SEE will have standard tools and procedures to record the interfaces. These standard tools will be for recording the products. They will not preclude the use of additional tools and procedures for the creative aspects of arriving at the definition of these baselined products. Such an approach allows variation and innovation in the creative aspects of the software engineering process while standardizing the form of the recorded products.

This document defines the six life cycle activities shown in Figure 1.4-1. It also describes the nature of the baselined products for which formal interface specifications will be developed. These interface specifications will be the subject of a future document.

1.5 Structure of the Framework

This Framework is divided into two major parts:

Part I describes a well-defined doctrine of software engineering disciplines, including the methodologies and activity-specific tools vital to it. This is

the foundation for the Navy SEE, as defined in Part II, and as it will evolve with experience and the advances of technology.

Part II defines the characteristics of a SEE as an integrated system consisting of a computer-based support system and data base.

Appendix A provides some background on the problems facing the Navy in software development. The SEEWG was a direct outgrowth of the forecasts summarized in Appendix A.

A glossary is provided to define the terms, phrases, and acronyms used within this document. It is not offered as a standard for terminology within the Navy or industry.

The heading for each page of the document guides the reader by telling them, to four levels of indexing, the part of the document being read. The page number tells the major section being read (where s is Part I, Part II, Appendix A., Glossary or Index) and the page number. Up to four levels of title will be given; the titles are either full or slightly shortened.

Page s-## Section a.b.c.d. Title "a": Title "b"
Title "c": Title "d"
****

PART I

CONTEXT OF

SOFTWARE ENGINEERING

Part I: Context of Software Engineering

Many people have had too narrow a view of a software engineering environment. It is seen as a set of automated tools supporting software development. Unfortunately, this approach cannot address the Navy's problems in software development. A software engineering environment must be seen as an integrated data base and support system; it will take a well-defined software engineering process and associated methodologies far more effective. With a comprehensive SEE supporting a well-defined process and methodologies, the Navy's software production groups should greatly improve their productivity as well as the reliability of their software.

Before the requirements for a SEE can be presented, the doctrine underlying a software engineering environment must be clearly defined. It would be unwise to design tools for building a house without understanding the process of building houses. Similarly, one should not try to design a SEE without first clearly defining the software engineering process. Therefore, Part I describes a comprehensive software engineering process and its associated methodologies so that a comprehensive SEE supporting it can be described in Part II.

Simply defined, a software engineering process is a set of activities for developing and modifying software through its life cycle. This process becomes the framework for a set of consistent software engineering methodologies. A methodology is a repeatable human procedure which supports some aspect of an activity. A well-conceived methodology separates the creative, intellectual aspects of an activity from the clerical, mechanical aspects. A tool is a computer-based aid which stimulates the creative, intellectual processes or automates the clerical processes.

Some may take exception to the methodologies selected but this is to be expected in any new field. These methodologies reflect good software engineering as it is known today and were chosen because they satisfy certain criteria:

- They improve the effectiveness and productivity of software development activities.

- They result in the creation of more reliable software.

- They fit together to form an integrated set of methodologies.

- They separate the creative aspects of software development from its more mechanical aspects.

- They promote the automation of the clerical, mechanical aspects of software development.

A good SEE requires a comprehensive set of methodologies. The implementors of a software engineering environment should include many methodologies, including some not covered in this document. However, the implementors should be guided in their selection by the criteria listed above.

1. Life Cycle Model

The life cycle model of a software system depicts the activities, and the relations among these activities, involved in the development, operation and life cycle support of the system from its conception to its retirement. Figure 1-1 provides a graphic representation of the life cycle of software (Fox [82]). In software engineering, the distinction between development and continuing adaptation (often referred to as maintenance) is arbitrary, depending when the first release begins operation. New releases, based on changes to the system, periodically upgrade the operational system.

The focus of the SEE description will be the development and continuing adaptation segments of the life cycle. Some operational requirements on the SEE, such as performance monitoring, will also be discussed. The following activity categories for software development have been selected:

Requirements Analysis - Definition and analysis of the user's evolving needs.

Specification - Translation of requirements into a precise description of the externals of the software system.

Design - Creation of an abstraction of a software system that is consistent with the specification

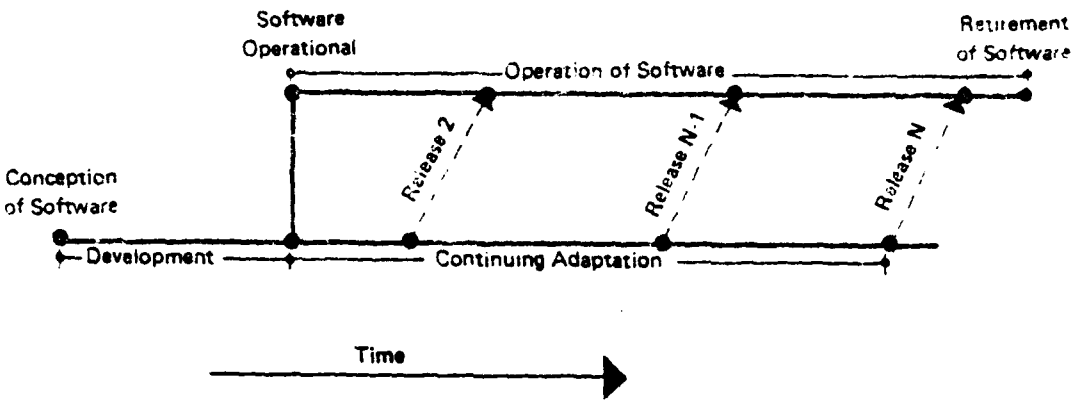


Figure 1-1: The Life Cycle of Software

and provides a reasonable description for implementation.

Implementation - Creating a software system which implements the design.

Correctness Analysis - Determining the correctness of each product of the software development process.

Management - Planning, organizing, operating, monitoring and controlling all activities within the software development process with the objective of meeting pre-established goals.

Although people have named and defined these categories differently, the real difference arises from the relations among the categories. These relations must be defined to achieve the best productivity possible while creating the most reliable software possible.

1.1 A Current Life Cycle Model

Figure 1-2 shows a simplified version of a common life cycle model. The activity categories shown do not match those defined in the previous section, but they map roughly as follows:

Activities	Current Model
Requirements Analysis	Requirements
Specification	Functional Design
Design	Detail Design
Implementation	Code and Unit Test Integration
Correctness Analysis	System Test
Management	(No match)

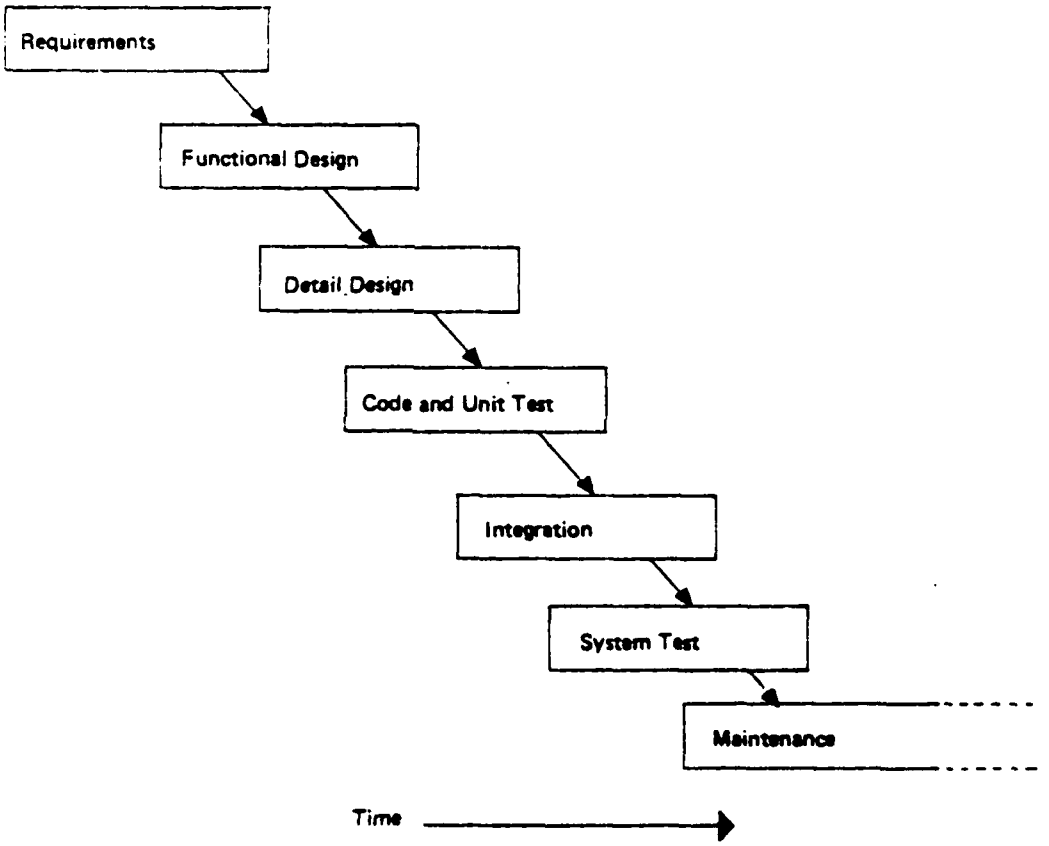


Figure 1-2: A Current Life Cycle Model

There are many versions of Figure 1-2, sometimes called the "waterfall" model. Some models use activity categories strictly tied to certain standards, others have verification feedback loops, but they show some or all of the following shortcomings:

Finite Activity Phases - Management assumes activities have distinct beginnings and endings. Although this notion is appealing to management, the actual technical work involves a continuing iteration of requirements analysis, specification, design, implementation, and correctness analysis over the life cycle of a software system. In a large project, the management view is imposed on the designers, thereby restricting the iterative process. Perhaps this is why a small design and implementation group is better at creating quality systems on schedule and within cost.

Static Requirements - Requirements are treated as though complete definition is possible early in the development phase, and as though no further changes will occur. Nothing could be farther from the truth. Requirements rarely are understood fully at the start of software development, and the evolving nature of the surrounding systems often lead to changes in requirements.

"Big Bang" Psychology - Software system development is seen as leading to the creation of a monolithic system. For large systems this requires coordinating the work of many programmers over extended periods with changing requirements. Such an approach has consistently resulted in overruns in schedule and budget and poor quality software.

Testing as the Primary Means of Correctness Analysis - Testing is the traditional way to perform correctness analysis. Because testing occurs late in the development phase, the majority of errors, which are introduced in the earlier activities, are very costly to correct (Boehm [76]). Correctness analysis, in addition to testing, must use other techniques to find and eliminate errors as they occur. Such an approach could greatly reduce software costs and improve reliability.

Development vs. Maintenance - Current life cycle models distinguish clearly between development and maintenance based on a date when the system [illegible] distinction between development and maintenance.

The Missing Management Link - Notably missing from the waterfall model is management.

1.2 A Recommended Life Cycle Model

Four ideas are recommended to remedy the shortcomings of current life cycle models:

- Incremental development
- Early prototyping
- Extended correctness analysis
- Management integration

Many development groups in industry have successfuly used some of these concepts. The effective use of all four could have a dramatic impact on software quality and productivity.

1.2.1 Incremental Development

Incremental development can reduce the disruptions on the developing system caused by changing requirements. It also increases management insight and control. The concept has been described in different forms (Mills [70]) and (Basili [75]). The basic idea is to build a software system in small, manageable increments, where each new increment is a new system with additional function. The first system release of a large project would consist of many increments.

Figure 1.2.1-1 shows a life cycle model based on incremental development. It is similar to the waterfall model shown in Figure 1-2 through requirements analysis, specification and design. Whereas the waterfall model continues into code, integration and test of the first release as a single production item, the incremental development model proceeds to successive increments. Each increment, as shown in Figure 1.2.1-2, is analogous to the complete waterfall model. Each reiterates requirements analysis, specification, and design to the degree necessitated by changes which occur at the base level since the last increment, or which results from the operational experience of previous increments. If no such changes have occurred, the increment consists of design elaboration, implementation, correctness analyses, and management only.

Using incremental development, it is not necessary to have a consistent complete set of software system specifications and design before the start of implementation. Because refinement and elaboration of these "front end" products are formally built into each increment, a natural framework for evolution is provided. Therefore it is necessary to perform initial requirements analysis, specification, and design to the degree sufficient to plan for development of Release 1 and to begin increment 1, but "complete" definition is not required until the last increment of Release 1.

By dividing each system release into several increments (6-8 months in duration), management of change is possible in a controlled, efficient manner. Estimation, which is usually based on metrics spanning the entire development phase can be calibrated at the end of each increment allowing for early detection and correction of schedule or resource changes.

In summary, incremental development provides the following benefits which are not possible with the current view of the software life cycle:

- It allows for early verification of estimates for development resource.
- It provides a framework for the non-disruptive evolution of requirements.
- It provides better insight into the system and application during development.

- It encourages better management insight and control.

Figure 1-1 shows the life cycle process for incremental development from a management perspective. That is, it depicts increments that are significant for resource allocation, tracking and control, quality assurance, and baseline[1] control. The designer's or programmer's view of a development, although constrained by the management view, has another dimension. Within an increment, a designer may go through many iterations of specification, high-level design, and detailed design before completing the work. This gives rise to two different sets of methodologies. One set supports the creative aspects, and the other records the result. We will return to a discussion of this important distinction in Section 2.

1.2.2 Early Prototyping

Early prototyping fits well in a life cycle model based on incremental development. A typical Embedded Computer System (ECS) software development activity might involve as few as three and as many as five or six increments during development of the first release. If the first increment is well designed and well planned, it can be tested as a prototype to evaluate ill-defined or risky requirements. In general, prototyping will require special tools.

Early prototyping can be enhanced by executable design and throwaway code. Executable design languages could shorten the time needed to produce a working prototype. Throwaway code is temporary code written to create some needed function for the earliest increments of the system. The use of throwaway code is like breadboarding in hardware development. It applies the Brooks' dictum: "plan to throw the first one away" (Brooks [75]).

---

[1] A baseline is a version of the software (and related information) which has been frozen so that subsequent changes can occur only through the controlled process of configuration management.

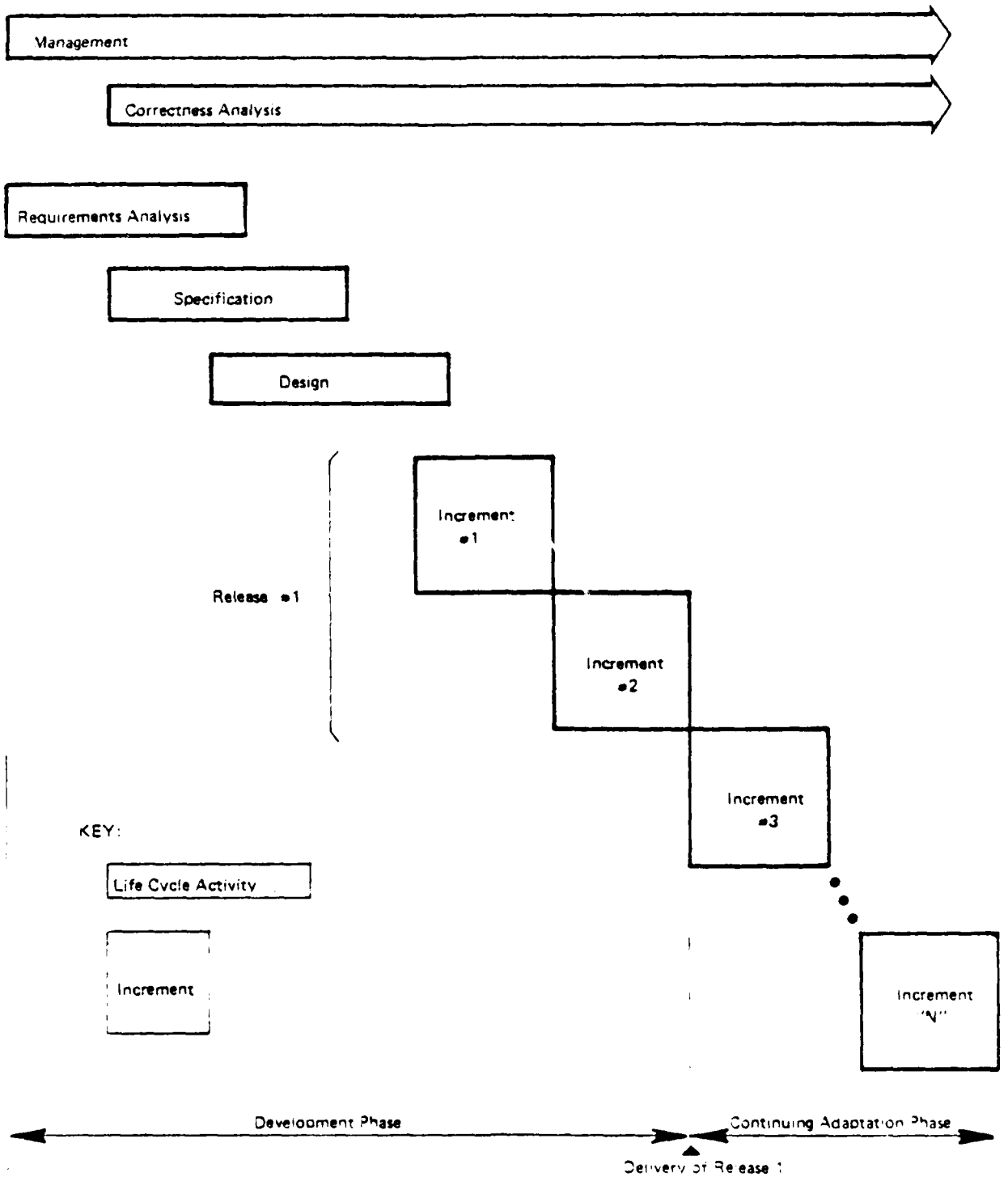


Figure 1-2 Incremental Development Model

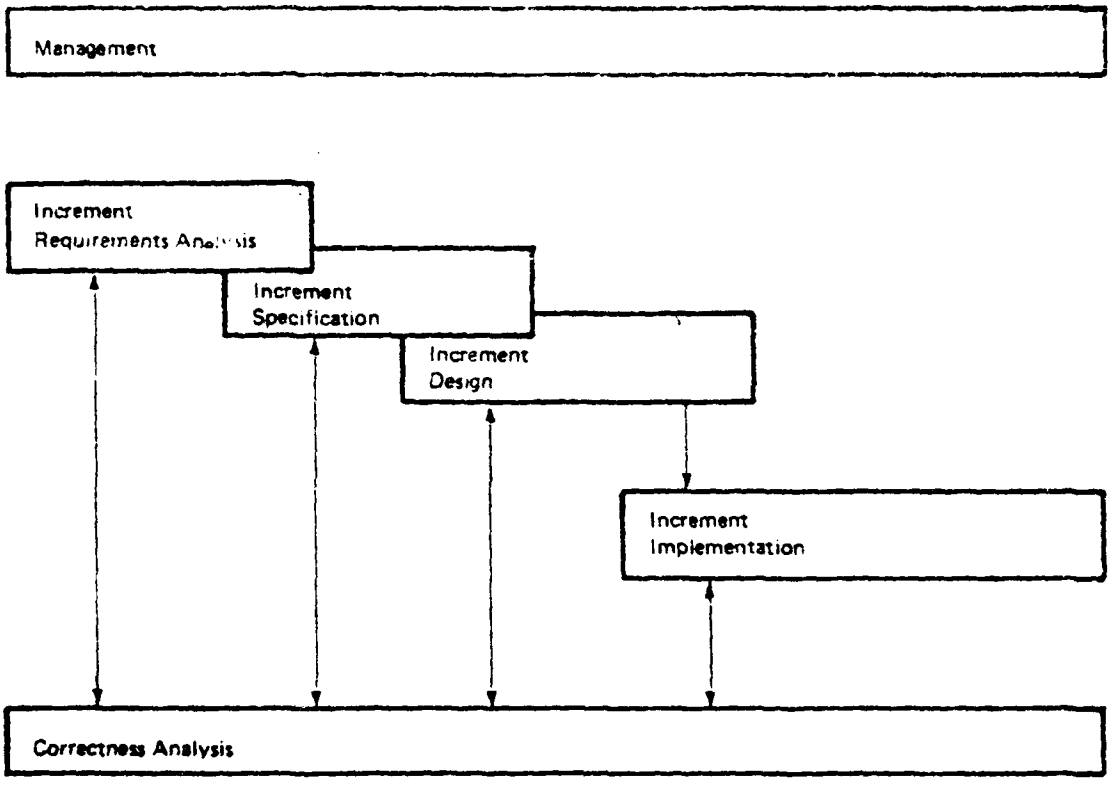


Figure 2.2: An Increment Sub-Model

1.2.3 Extended Correctness Analysis

Figures 1.2.1-1 and 1.2.1-2 show correctness analysis extending throughout development. Studies have shown that 50-80% of all errors are introduced during requirements analysis, specification and design (Boehm [76]). Since it may cost up to 100 times more to remove errors during testing than shortly after introduction, extended correctness analysis is sorely needed.

To achieve such savings, techniques must be used to show that the specification correctly implements the requirements, that the design correctly implements the specification, etc. Thus, correctness analysis spans all activities.

1.2.4 Management Integration

Management is an important activity often omitted from life cycle models. It provides control over other activities and provides contingency plans for unexpected events. To be effective, it must be integrated closely with all other activities.

Management involves both insight and measurement. Insight involves seeing what people are doing as it is being done. Measurements are needed to evaluate the work and to allow management to learn from past efforts.

For instance, the frequency distribution of errors over a development activity, the average time to detection, and correlation of error types to methodologies used, could lead to improved methods of software development. Measurements also allow management to assess progress and to make adjustments for unplanned situtations.

1.3 Relation to Acquisition Management Models

The life cycle model presented in Section 1.2 supports the definition of requirements for a SEE. More specifically, it supports a discussion of the software engineering process contained within a SEE. This model does not match directly

with other Navy or DOD life cycle models used to define the acquisition process. Figure 1.3-1 shows a system acquisition life cycle model currently under consideration as a [illegible] whether the software life cycle model of [illegible] be mapped to the acquisition model of Figure 1.3-1, for the SEE must fit well within the Navy system acquisition framework.

The software incremental development model maps to the acquisition model activities bracketed between system concepts and system integration and testing as follows:

Acquisition Model	Incremental Development Model
System/Software Requirements Analysis	Requirements Analysis
Software Requirements Analysis	Specification
Preliminary Design & Detailed Design	Design
Code, Unit Testing, and Integration Testing	Implementation
Software Performance Testing Formal Reviews (e.g., PDR)	Correctness Analysis
(No equivalent)	Management

The incremental development model activity of design is divisible into system level design and program/data design (see Section 2.3). Therefore it matches up well with the corresponding activities of the acquisition model. The correctness analysis activity of the incremental development model addresses both reviews and testing activities. [illegible] includes unit, integration, and performance [illegible] as well as the formal reviews - PDR, CDR, etc.

The acquisition model requirement for [illegible] reviews such as Preliminary Design Review (PDR) and Critical Design

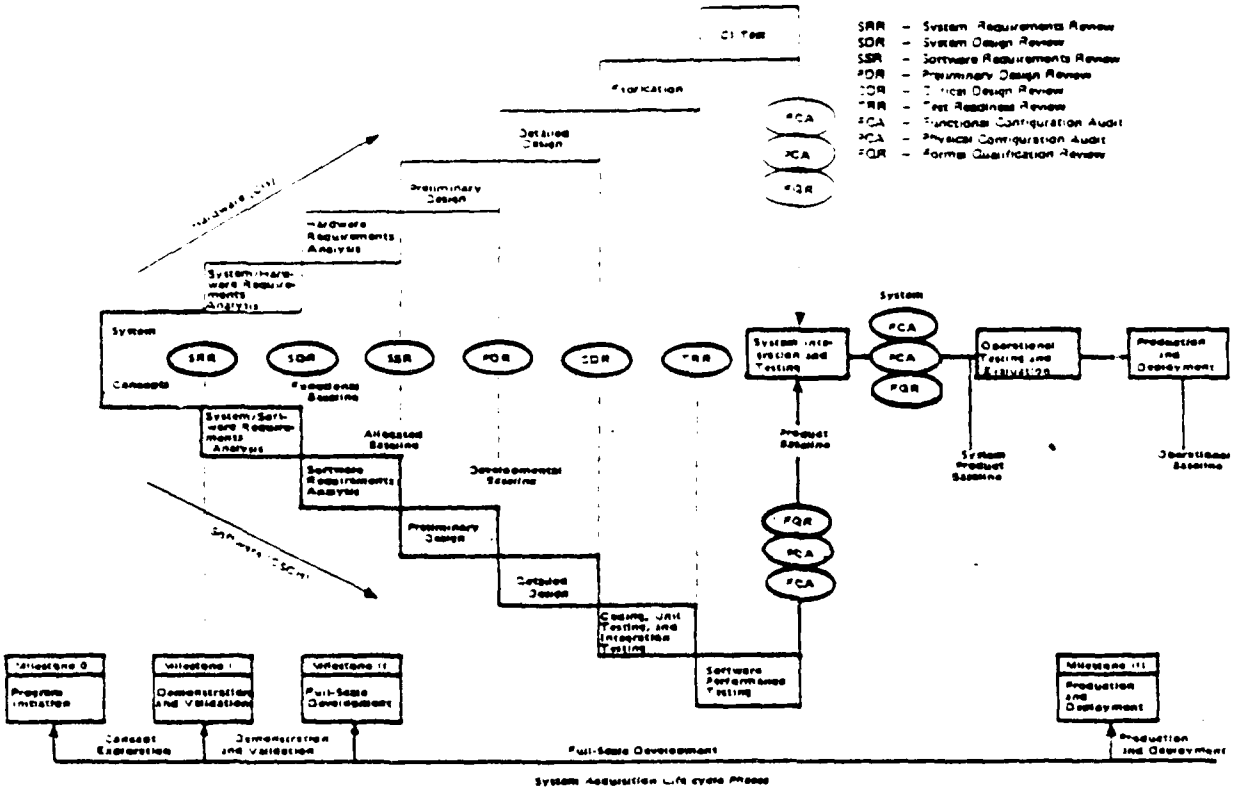


Figure 1.3-1: Joint Logistics Commanders Proposed Life Cycle Acquisition Model [1]

[1] From draft joint policy document titled "Management of Computer Resources in Defense Systems," 9 October 1981

Review (CDR), can be accommodated within the incremental development model with some minor adjustments. Software Specification Review (SSR) would occur at the end of the first system specification. PDR would occur at the end of the first design activity within each increment, and CDR would occur at the end of the design activity within each increment. With incremental development, multiple CDRs, one in each increment, will be needed. Test Readings Review (TRR) would occur when the system testing of the final increment in a release is complete. System Requirements Review (SRR) and System Design Review (SDR) would occur within the wider perview of the ECS development management.

1.4 SEE Requirements Derived from the Life Cycle Model

As stated in the introduction to Part I, the [illegible] the software engineering process and the methodologies to be supported largely determines the SEE requirements.

The support system and data base of the SEE must fit the life cycle model based on incremental development. Incremental development levies several important requirements on the SEE:

- The data base must support the baselined products by increment, allowing for the possibility of increments overlapped in time.
- The data base must also allow programmers to keep intermediate products in their work space, providing for iteration within increment.
- There should be an integrated set of tools used to define the products in the baseline, and tool sets to aid the designer/programmer while working toward the baselined products.
- Methodologies and tools, in addition to testing, must be provided to ex[illegible] correctness [illegible] to all life cycle activities.
- Data pertaining to these methodologies and tools must be captured in the data base.

- Measurements of each activity must be captured for management.

- Tools for drawing measurements from the data base and for analyzing the measurements are also needed.

2. Methodologies and Tools

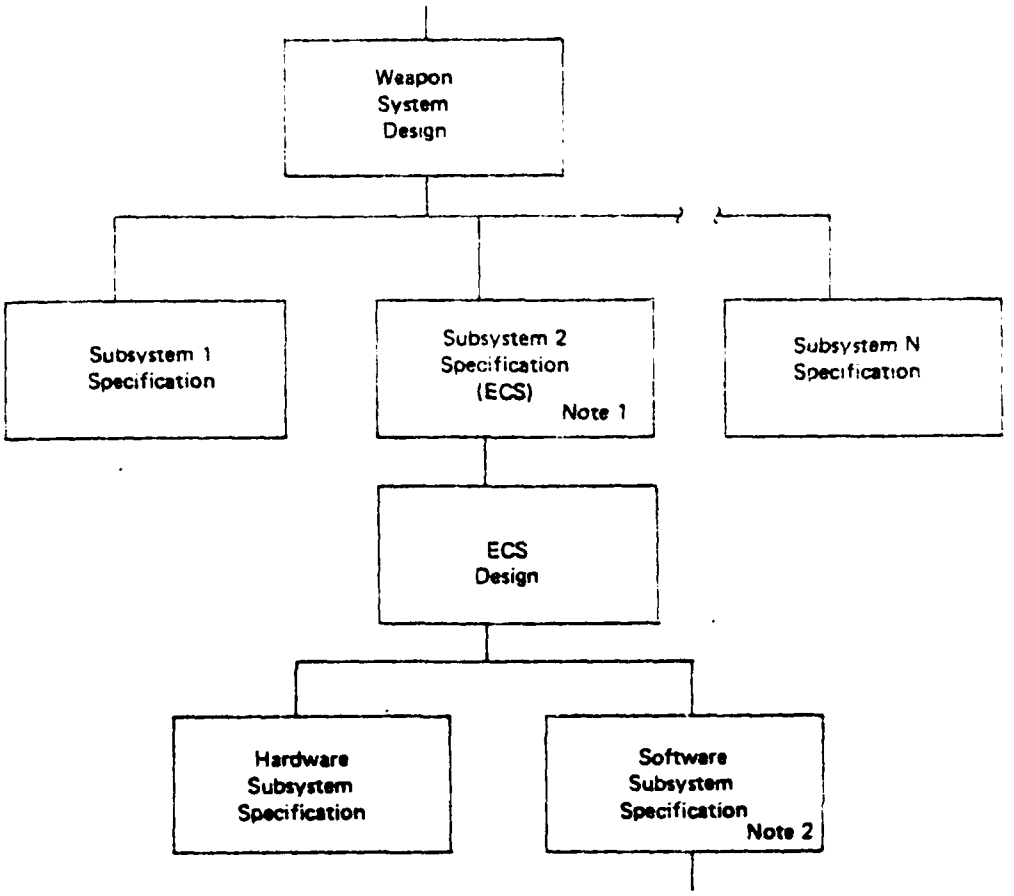
The following subsections explore methodologies and tools which support the software engineering process defined by the incremental development life cycle model. The subsections also summarize the requirements they levy on the SEE. The methodologies chosen reflect today's state-of-the-art. Certain selected alternate methodologies might be chosen without major changes to the SEE requirements.

For each life cycle activity, the activity is defined and characterized and applicable methodologies and tools are then discussed. The methodologies and tools are categorized as follows:

> Formal Recording Aids - Those methodologies and tools associated with the capture of baseline products. They must fit together from one activity to another and promote traceability, which is the relation of products (or their parts) in one activity to the products (or their parts) in adjacent activities in the life cycle model.
>
> Creativity Aids - Methodologies or tools augmenting the creative, intellectual processes.

A minimum set of methodologies and tools for formal recording is required. The creative aids, on the other hand, can be anything that helps an analyst, designer, programmer, or tester, create a product. The product is then formally recorded for baseline control under configuration management.

The life cycle model described in Section 1.2 provides a framework for defining appropriate methodologies and tools. However, it must be placed in the context of the weapon system acquisition life cycle for an understanding of the role of the front-end software engineering methodologies. Figure 2-1 shows a progression of weapon system



NOTE:

1. **IDEAL POINT OF FIRST INVOLVEMENT BY SOFTWARE DEVELOPMENT (REQUIREMENTS ANALYSIS).**

2. **USUAL STARTING POINT FOR SOFTWARE DEVELOPMENT (SPECIFICATION).**

Figure 2 1: The Starting Point of the Software Engineering Process within a Weapon System Life Cycle

design and specification[1] documents at some point after the definition of the weapon system concept. In an ideal world, the ECS software development agent would first enter the process at the ECS Specification and Design (Type A specification) to ensure feasibility of the approach. The formal starting point for software development would be the Software Subsystem Specification (Type B specification).

The requirements analysis activity, which is the first activity in the incremental development model, of necessity will involve the software development agent in system level questions such as hardware/software trade-off issues. Hopefully, such involvement will lead to a set of system level requirements which will allow efficient design and implementation of quality software.

2.1 Requirements Analysis

Requirements analysis involves looking at the requirements within the ECS specification and design level before developing the software system specification.

In most cases today, the progression of Figure 2-1 is just an ideal. More likely, the software development agent is brought in after the equivalent of the ECS design is complete. Furthermore, the requirements for the software, as represented by the ECS specification and design documents in the ideal case, are not well defined. They may be contained in many sources including documents, memos, and the minds of the ECS designers. Therefore, requirements analysis also must address the capture and interpretation of requirements from these sources.

Although requirements analysis begins the software engineering process, this activity does not necessarily end with the start of software specification. Requirements analysis continues over the software life cycle because of

---

[1] Specification was defined in Section 1 as a rigorous external description of a system without concern for its internal workings. This type of description is sometimes referred to a "black box."

changing requirements resulting from engineering change proposals (ECPs) and trouble reports (TRs).

2.1.1 Characteristics

Requirements are binding conditions. In Figure 2-1, requirements on implementors at one level in the hierarchy are any binding conditions imposed at the higher levels. These requirements can take many forms. For the following discussion of requirements analysis, the different forms of requirements will be classified as follows:

> Context - Bounds on alternatives for the software system design implied by the system mission or environment.
>
> Description - Characterization of what the required system is to do without concern for how it will work.
>
> Constraints - Any conditions which must be met in designing or implementing the required system or any quantitative performance measures to be met by the operational system.
>
> Evaluation Criteria - Rules to be used in judging the quality of the completed system.

Requirements analysis involves two sub-activities: requirements interpretation and feasibility assessment. Interpretation concerns identifying and defining software requirements so that correct specifications are possible. Feasibility assessment concerns determining the reasonableness of implementing a system that can satisfy the requirements within schedule and cost. The identification of risk in this regard may lead to a redefinition of the requirements with the ECS designers.

2.1.1.1 Requirements Interpretation

Semantic models provide the best method for interpreting the context and description of the requirements. A semantic model consists of categories for terms and relations among

categorized terms. Examples[1] of techniques based on semantic models are Structured Analysis and Design Technique (SADT) (Ross and Shoman [77]) and Problem Statement Language (PSL) (Teichroew and Hershey [77]). The underlying semantic models of SADT and PSL differ in categories and relations chosen.

There are several benefits to expressing the requirements in the form of a semantic model:

Definition in Context - Key terms of the requirements, defined in the context of other problem-related terms, reduces ambiguity and increases understanding between the weapon system engineers and the software engineers. Thus, semantics information can be thought of as a sophisticated data dictionary.

Specification Mapping - If the semantic model categories and relations were properly chosen, the semantic information will suggest a natural mapping for the system specification. This specification precisely expresses the description requirement.

Consistency/Completeness Analysis - The semantic information depicting the context and description requirements can be analyzed for both consistency and completeness. This analysis is based on the relations found in the semantic model.

Traceability - The semantic information can establish traceability between the requirements source and the specification. It does this by including references in the definitions of terms. Later, these references can be extended to related baselined products.

Design Aid - Naming consistency can be maintained, and traceability can be extended, by definition of

---

[1] Throughout Part I of this document, existing methodologies and tools will be cited to illustrate the feasibility of the concepts presented. They are cited as examples only. There is no implication intended that these be taken as requirements on a Navy SEE.

[illegible] during specification and design. The semantic information can also serve as a [illegible] for data base design.

2.1.1.2 Feasibility Assessment

Feasibility assessment determines how reasonable the constraints and evaluation criteria of requirements are. For complex systems, feasibility assessment usually requires a trial design first. Such a design demonstrates the feasibility of the context, description, and constraint requirements (but may overlook other considerations). A model or simulation of a trial design can be used to show how it would perform if implemented. Modeling is an excellent technique for assessing the feasibility of performance constraints.

Resource estimation techniques are useful. They can assess the feasibility of implementation constraints such as a milestone of initial operational capability. They can also assess the impact of evaluation criteria. Applying these techniques to the trial design (allowing for extra work to meet the evaluation criteria), one can estimate reasonable resource and schedule expectations. These, in turn, can be compared to the implementation constraints.

Risk areas can be identified with modeling and resource estimation techniques. The requirements might then be changed to eliminate infeasible or conflicting requirements, or to reduce the risk linked to certain requirements.

2.1.2 Methodologies

Several methodologies can be used during requirements interpretation and feasibility assessment.

> Semantics Information Capture - Supporting interpretation, the capture of requirements in the form of a semantic model involves identifying key terms, categorizing the terms, defining the terms, and identifying the relations between the terms. The capture of semantics information creates a formal recording of the semantic model of the

requirements, which becomes part of the baseline. An example of a comprehensive methodology for semantics information capture is suggested in Wilson [79]. Assuming the semantics information is machine-encoded, it might be expressed in a formal language such as Problem Statement Language (PSL).

Semantic Analysis - Once the requirements are expressed in the context of a semantic model, the model relations can be used for a systematic analysis of the completeness and consistency of the requirements. This is achieved by asking questions which are answered with the aid of the relations, such as "Are there any other processes which should be related to Process A by the 'predecessor of' relation?"

Traceability may be established through reference relations between requirements and specification, design and code, etc. The relational analysis can be used to assess the impact of requirements changes on the baselined products.

The semantic analysis methodology also aids creation by identifying areas of requirements incompleteness or inconsistency.

Feasibility Analysis - Evaluating the feasibility of requirements is a significant part of requirements analysis. Feasibility should be viewed from the perspectives of design, performance and cost.

Design feasibility involves finding at least one design that satisfies the requirements. Any approach from trial design to prototyping is appropriate. Performance feasibility is a special case of design feasibility analysis. Once a trial design is established, modeling is an effective technique for analyzing performance. Cost feasibility involves estimating costs based on the trial design. Cost analysis must consider the three key elements: the development phase, the operations phase, and the phase for continuing adaptation.

The software development agent may have to go through several iterations with the weapon system or ECS manager. Together they can resolve problems discovered during

requirements analysis, such as infeasible, inconsistent, or over-constraining requirements. The methodologies summarized above may be used before an acceptable set of requirements for the software system is reached.

2.1.3 Supporting Tools

Several tools are effective in supporting the methodologies described above.

Semantics Language Processor - A language processor could perform syntax checking and recording of semantics information entered via a language such as PSL. The results would be available for later analysis or retrieval. The information would resemble a data dictionary augmented by relations.

Information Storage and Retrieval - This tool would store and retrieve the semantic information. It might be based on a data base management system (DBMS) using relational techniques. An example of such a tool used in this application is IBM's Query-by-Example (QBE) (Perriens [79]).

Semantics Information Analyzer - This tool would use the relational nature of the semantics information to perform consistency and completeness checks. Problem Statement Analyzer (PSA) (Teichroew [77]) is an example of a tool used to analyze data captured through the use of PSL.

Report Generator - General report generators would help create requirements analysis reports. Examples of reports are (Orzech [79]):

- Formatted problem statement report, which gives the original relationships in a well-structured format.
- Structure report, which presents relationships as a hierarchy.
- Data structure report, which lists data structures and the types of their contents.

- Data/activity interaction, which shows interaction between data objects and activities, and

- Picture report, which diagrams the direct relationships of an object.

Modeling Tool - This tool would provide queuing theory aids tailored to descriptions of computer systems. The tool assists in developing performance analysis models. Performance Oriented Design (BGS [78]) is an example of such a queuing tool. Other modeling tools besides queuing would also be appropriate.

Resource Estimator - An estimation model can help assess cost feasibility. The model would use characteristics of the software system and the development approach to predict required manpower, schedule, and computer resources.

Prototyping - Tools for developing prototypes of software systems would be useful. They might consist of a high-level language and an interpreter.

2.1.4 Requirements on the SEE

The requirements on the SEE data base, derived from requirements analysis, are:

Baselined Products - The semantics information is the only data associated with requirements analysis that should be baselined. It should be under configuration control and subject to change only as requirements changes are approved.

Non-Baselined Data - Any information associated with modeling, simulation, prototyping, or semantic analysis should be saved temporarily. It can be used later in requirements analysis iteration or other activities.

Measurement Data - Several measurements of the requirements analysis activity and its outputs should be captured:

- Size of the data base for semantics information,
- Complexity of the requirements as measured by the relationships in the semantics information, and
- Number of inconsistencies or omissions found.

The support system must provide a framework for the types of tools summarized in Section 2.1.3. Given the relational nature of the semantics information, another requirement might be that the DBMS support relational operations.

2.2 Specification

The specification activity should transform the descriptive requirements into a precise statement of the software system external behavior. Two key information sources are used in this transformation: (1) semantics information captured during the requirements analysis, and (2) the referenced requirements information from which it was derived. The resulting specification, along with the constraints and evaluation criteria requirements, provides the basis for design, implementation, and correctness analysis. In most cases the constraint requirements should be included as part of the specification (perhaps as an appendix). Thus, the specification will contain all that must be known to implement the software system. The specification also provides the basis for judging the correctness of the implemented system.

2.2.1 Characteristics

Specification development consists of two distinct acts. The first is to collect information that the specification will contain, this is a creative process. The second is the act of recording this information. Although these actions may be carried out simultaneously, normally in an iterative fashion, they should be considered separately since each suggests its own methodologies and tools.

Developing the system specification must be done in such a way that the specification can always be shown as complete and consistent.

The semantics information produced during requirements analysis can provide traceability between the system requirements and specification. This link can show subjectively the correctness of the specification. It is helpful in providing insight into how the specification must be updated as changes to the requirements are made.

If feasibility analyses were not conducted during requirements analysis, they should be conducted during specification development. Again, this analysis consists of constructing a model (or simulation of the system) and performing analyses to see if the desired timing and accuracy criteria can be met.

2.2.2 Methodologies

The specification information must be recorded in some suitable form. As a minimum, the specification should describe:

> Interactions - the interactions of the software system and its outside environment (i.e., descriptions of I/O device interfaces, sensor interfaces, etc.);
>
> Modes - externally visible modes of operation (e.g., "initialize" or "auto pilot"); and
>
> Functions - externally visible functions (e.g., mappings of inputs to outputs).

Furthermore, any methodology for recording the system specification should produce a document which is:

> Minimal - The purpose of a specification is to define what the system must do. A minimal specification will describe only this; and it will not imply any special use of the system. Such specifications are said to be black box because they describe only the externally observable behavior of the system. A minimal specification has the

advantage of not restricting the system designer to the point of carrying out an implementation that [illegible]

Understandable - Since the specification will be used heavily as a reference tool by system implementors and maintainers, it should be clear and concise. The information must be well organized so that answers to particular questions are easy to find. The text should be free of jargon.

Accurate and Precise - The specification must be as complete and consistent as possible. The specification should use as much formal notation as possible. Such notation is needed for a formal correctness analysis. It also makes automated analysis of the specification information easier.

Easily Modified - The evolving nature of requirements suggests that the specification information be easy to update and change. This can be done by organizing the information so that different aspects of the specification are described separately and independently. This approach minimizes the impact of a single change to the specification.

The creative aspect of specification development is the synthesis of the proper information for the specification. This necessarily involves:

Completeness Analysis - This is done by trying out a design of the system. Completeness analysis generates questions which can help identify information absent from the requirements. In most cases, this activity will be done during requirements analysis.

Correctness Demonstration - The specification must be shown as consistent with the requirements. Since the requirements may not be stated in a formal manner, a rigorous proof of their consistency may not be possible. The correctness demonstration is then produced through a subjective, informal analysis based on the semantics information from requirements analysis (Ferrentino and Mills [77]).

Consistency Analysis - Any methodology for performing this analysis depends on the form of the specification information. If a formal specification language is used, certain kinds of problems may be detected by analyzing this notation. In other cases, the consistency of the specification information must be judged on a less precise basis.

A good example of the state-of-the-art in specification methodologies is that advocated by Parnas and Heninger [18], and used to develop the specification for the A-7 flight program. The specification document includes formal, tabular notation which lends itself to completeness and consistency analyses.

2.2.3 Supporting Tools

The following tools help to develop the specification.

Language Analyzers - These can check for errors in any formal notation within the specification information. These analyzers are tailored to the specification language or notation used. They receive, as input, the specification (or some portion thereof) and issue diagnostics concerning syntax errors.

Consistency/Completeness Analyzer - These analyze information entered via formal language and detect inconsistencies or omissions in the specification. For example, tables might specify some outcome as a function of various factors. The tables can be checked to ensure that an outcome is given for every event, and that no event corresponds to two mutually exclusive outcomes.

Modeling Tools - These may be needed if no performance feasibility study was conducted during requirements analysis.

Report Generators - These are needed to produce a finished document from the specification information stored in the data base.

2.2.4 Requirements on the SEE

The requirements on the SEE data base, derived from the specification activity, are:

Baselined Products - The specification information is baselined. Any modeling information produced might be baselined if it is crucial to the life cycle support of the software.

Non-Baselined Data - This material includes partial specifications under development, alternate specifications, and diagnostic information produced by specification analysis tools.

Measurements - Examples of useful measurements are: effort and resource data concerning the development of the specification, size data, number of errors and changes made, and subjective measures of the quality and completeness of the specification.

2.3 Design

Design involves creating an abstraction of a system. When implemented according to this design, the system will meet the software system specification, the constraint requirements, and the evaluation criteria. A design formally describes the system. Decomposition and abstraction are the key means of dealing with the complexities of a system during its design. Decomposition involves dividing the system into pieces (modules, programs and data structures), which can be related in some hierarchic fashion. These pieces can be handled independently during further decomposition of the design. Abstraction is used to suppress all but those details needed to understand the design at any level.

The concept of hierarchy is important for traceability and correctness of the design. The correctness techniques developed thus far rely on a hierarchy in the design. Furthermore, a complex design can be better understood through abstraction plus a hierarchy showing the relations among the pieces.

A precise specification of the system is vital to the design process. Design can begin without complete specifications, but the missing parts must be clearly acknowledged as unknown. The specification must be complete by the time design is complete.

2.3.1 Characteristics

Design consists of creation and recording. Creation is a highly intellectual process. Several alternative designs may be created and assessed before a design approach is chosen. The creation activity draws on the skill and experience of the designers. It may require modeling and trade-off studies to reach a reasonable solution. Once a design approach is chosen, the formal recording act depends on the means of decomposition and choice of abstractions. Formal recording is enhanced by techniques which foster clarity, understanding, correctness analysis, and ease of change.

Another view of design highlights the difference between system design (design-in-the-large) and data and program design (design-in-the-small). System design involves breaking down (decomposing) the software system into hierarchically-related modules. Each module, in turn, has a module specification. Design-in-the-small addresses the design of individual modules. It defines abstract data structures and programs composing the module. These programs, in turn, may invoke functions of other modules. One way program designs may be described is through a Program Design Language (PDL)[1].

The method of decomposing the software system into modules is central to software system design. If modularity is not viewed properly, the software will not be easy to change.

Two related concepts that are important for creating good modules are information-hiding and data-intensive design. Information-hiding isolates design decisions likely to change in the future. It also provides an interface

---

[1] PDL is also used as an acronym for "process design language" or "procedure design language."

between the module and the rest of the product that remains valid for all versions. Module specifications designed with information-hiding would reveal functional capabilities, not the design decisions underlying the implementation.

Many people view data rather than function as the major factor influencing a design. The design decisions deal with data--its structure, the means of access to it, and the factors involved in altering it. Module design, using the concept of data-intensive design, would begin with the creation of abstract data structures, followed by the design of the programs acting on these data structures.

In summary, the recorded design consists of the following elements:

- Module hierarchy representing the system decomposition,
- Module specifications to reveal the external descriptions of the module, and
- Module designs consisting of program designs and abstract data definitions.

The design of the system must support incremental development. The design's decomposition structure should allow the system to be built incrementally. Such a structure may make the implementation easier and will almost certainly make continuing adaptation easier. It also allows the designer to focus on the more difficult parts of the design in the beginning, thereby improving the chances for the best solutions.

2.3.2 Methodologies

Three groups of design methodologies are identified: formal recording of system design, formal recording of data and program design, and creative aids.

There are several techniques for recording the system design.

Information-Hiding - This technique involves isolating information within modules. The module

limits are defined by the information (design decisions, data definitions, etc.) to be isolated. Design is based on the expected changes to the information, thus localizing the effect of future changes (Parnas [79]).

Module Specification - This technique allows others to determine the intent of a complete module by reading the module specification.

Uses Hierarchy - This technique explains which programs depend on the correct implementation of a given module to produce correct results (Parnas [79]).

The techniques for the formal recording of data design and program design are:

Program Design Language (PDL) - PDL is a useful technique for formally recording the program design. It is sufficiently low-level to support direct coding, and is flexible enough to leave some questions unanswered while the designer proceeds with the design.

Stepwise Refinement - This technique goes hand in hand with PDL. With stepwise refinement, specifications for the lower level code become part of the documentation of the procedure. This makes the intent of the code much clearer.

Abstraction of Data Types - With abstraction, the designer can develop details where they are needed (Guttag and Liskov [77]). This permits information-hiding as well as a more independent implementation of the system.

Many creative techniques exist for design. A designer chooses techniques based on their individual approach to creativity. Some prefer graphic techniques while others do not. The choice of creative techniques should be left to the individual, whereas the techniques for formal recording must be standard. Described below are some representative creative aids:

Data Flow Analysis - Module decomposition and function allocation are based upon the data flows required by the system. An example is Structured Design (Stevens [74]).

Data Structure Transformation - Transformation is a design technique in which the structure of the input and output data determines the structure of the program (Jackson [77]).

Graphic Decomposition Techniques - Graphs showing hierarchic relations depict the decomposition at many levels. An example is Structured Analysis and Design Technique (SADT).

Graphic Control Descriptions - Other ways of showing the control flows in the program are Petri Nets and Warnier-Orr diagrams.

2.3.3 Supporting Tools

Figure 2.3.3-1 lists the design methodologies outlined in Section 2.3.2 and identifies tools that might support some of the methodologies. The tools are briefly described below.

Design/Specification Language Processor - These check syntax and connections. Input is the system design and the module specifications written in some standard syntax. Output is a list of inconsistencies or syntax errors.

PDL Syntax Analyzer - These detect mismatched interface items and force the designers to maintain a consistent syntax for the design. Input is a design written in a PDL, and output is a list of syntax errors and inconsistencies in data usage.

PDL Interpreter - These would allow a design to be executed before it is actually coded. The interpreter's input is a design written in a PDL with program input data. Its output shows the result of applying the input data to the design.

Graphics Package - These would support the creative phase of design. Various designs could be displayed graphically to aid the designer in making a choice. Input might be module descriptions, hierarchy relations, or data-use relations. Output might be graphic representations of this information.

Methodologies	Tools
Formal Recording:	
Information-Hiding	
Module Specification	Design/Specification Language Processor
Uses Hierarchy	
Program Design Language	PDL Syntax Analyzer PDL Interpreter
Stepwise Refinement	
Abstract Data Types	Report Generator
Creative Aids:	
Data Flow Analysis	Graphics Package
Data Structure Transformation	Modeling Tools
Graphic Decomposition	
Graphic Control Descriptions	

Figure 2.3.3-1: Methodologies and Tools of Design

Modeling Tools - These are useful for judging [illegible] feasible particular designs are. A model might take a high-level design as input. It might then produce execution and timing statistics. These statistics may be used to determine if the design could meet the performance requirements.

Report Generators - These transform stored design into documents and reports. Thus the baselined design will be stored in machine-readable form, permitting required documents to be produced easily.

2.3.4 Requirements on the SEE

As with the other activities of development, the data base must contain information on the design.

Baselined Products - Throughout the life of the system, the most recently approved form of the design must be stored in the data base. The system design may be entered before the design of various subsystems or modules.

Non-Baselined Data - This includes preliminary designs as well as graphic displays used during the creative process. Graphic displays may include tree structures, block diagrams, and other material created by design tools. The data base must provide for maintaining the temporary designs developed before one is actually chosen and baselined.

Measurements - These might include module interconnection measurements, such as data bindings (Basili and Turner [75]). These might also include lower design measurements, such as cyclomatic complexity (McCabe [76]), and operators and operands (Halstead [77]). Many of these measurements are normally taken on the completed code, but with good, low-level PDL, they can be taken (or approximated) during design.

Archival Data - Archived data should capture the motivation behind the choice of design. The archived data should also include past designs evolved from use or rejected during development along with the reasons for the rejection.

2.4 Implementation

The implementation phase should create a system which implements the design correctly and meets the resource, accuracy, and performance constraints in the specification. Implementation translates design into the language of the target computer, the data base, and other products needed to create the operational system.

2.4.1 Characteristics

Implementation should produce an effective, understandable realization of the design. Clear, readable source text is required for ease of maintenance.

The following are the key aspects of implementation:

Code Generation - This translates the design to the machine language of the target computer. It varies in difficulty based on the level of the design and the level of the target language. Code generation begins with the expansion of the detailed design into the appropriate programming language. As with system design, correctness arguments make a correct implementation easier to produce.

Assembly/Compilation - The generated code must be compiled or assembled. This process checks the code for certain errors and produces its relocatable binary translation (or object module).

Debugging - It is usually helpful to do as much debugging as possible on the host computer rather than on the target computer. This is especially true of unit testing. The testing might be done on the development host using a simulator of the target machine. In some cases, a host-to-target connection is used for the test. Another option is to translate the source code for direct execution on the host computer.

System Integration - Units from the compilation activity are tagged with information on increment, version and revision number, and are stored in the

project data base. When the implemented system is ready to be built, software generation tools (either the compiler or linker/loader) select the proper units from the data base. The tools then may recompile those which need it and diagnose incompatibilities across units. (These concepts of integration differ from those of the Ada development process.)

System Generation or SYSGEN - This process involves creating an operational system for a target computer. It uses integrated components plus instructions to the operating system, the linker/loader, the data base management system, and special SYSGEN tools. The output from these instructions are load modules, a data base or files, and the structures that define the system configuration. These will then be recorded on a transportable physical medium such as magnetic tape. They will be in a form suitable for loading the system into the target ECS.

Optimization - The system, or parts of it, may need to be "fine tuned" to meet performance specifications. The performance criteria will vary according to the system and can include external criteria such as response times for transactions, or internal criteria such as memory limitations, "thrashing" by the virtual operating system, or priority structures.

The programming environment must be able to support Ada as well as other Navy standard languages or authorized non-standard languages. Supporting Ada requires: a compiler, a run-time environment capable of the tasking and exception-handling features of Ada, and a mechanism to handle the separate compilation facilities of the language.

2.4.2 Methodologies

In many ways, translating a design into a system for the target computer is a straightforward (indeed, almost mechanical) task. Thus few methodologies and many software tools are available for this aspect of software development.

Structured Programming - One-in, one-out control structures are used for all programs; thus, programs are also one-in, one-out. Programs are viewed as

function composition of the one-in, one-out control structures providing a rigorous basis for correctness analysis.

Top-Down Implementation - System components can be designed, implemented and integrated in such an order that, throughout the integration process, an operable subset of the system will exist; this is called top-down implementation. As implementation proceeds, each new program is integrated into the operable subset. Top-down implementation is like incremental development in-the-small. In fact, it is performed within development increments.

Programming Teams - These have proved to be an effective means of increasing productivity and quality (Baker [75] and Weinberg [71]). On large projects, the system design must be decomposed. Then components can be assigned to several teams for concurrent development with a minimum of interaction among the teams. Productivity and reliability normally improve when interactions are minimized.

Instrumented Analysis - Errors can be identified, located, and removed with data taken from the operational system during execution.

Parameterization - Well-designed systems have many system-defining values that need to be changed as the system evolves through its life cycle. These values (or parameters) define such items as the number and size of I/O buffers, the code designated as permanently resident in memory, and screen formating data. Parameters can be either single values or tables of values. Parameters in a well-designed system can be changed at SYSGEN time or at start-up time without modifying any of the code in the system.

2.4.3 Supporting Tools

The following tools facilitate implementation.

Compiler - Reliable compilers are required for high-level language coding. For "unstructured" programming languages, such as FORTRAN, a preprocessor

... compiler permits the implementor to ... structured programming directly. DOD and Navy ... required in progress such as Ada and ... with ... capability.

Assembler - If the software is to be implemented in assembly language, an assembler with a macro facility is essential. Navy standard military computers must be supported.

Linkers and Loaders - These programs resolve external references among separately compiled or assembled programs, and create a "load module" for execution on the target machine.

Simulators - Object modules for the target machine can be tested on the host computer with simulators. Simulators for Navy standard military computers are a minimum requirement.

Host-Targeted Compilers - An alternative to simulators are host-targeted compilers. These tools use the source text created for the target computer and generate object modules which can be run directly on the host computer.

Debugger - Debugging is easier if it can be done using source text rather than object modules. The debugger executes the source text instruction by instruction. When an error is found, the debugger translates the state of the computer into information useful to the programmer, e.g., line number of the source text where the error occurred, values of variables, input data being processed, etc. The debugger also allows the programmer to set break points in the source text. These points suspend execution, thereby allowing the programmer to examine (or modify) variables before execution is resumed.

Pretty Printer - This program reformats the source text into a standard layout which makes the source text easier to read.

Analysis Tools - These include programs which produce cross reference listings, maps of variables which are being set and used, analysis diagnostics of data and control flow, as well as various measures of

size, complexity, performance, etc. These capabilities may be included as options in the compiler.

Data Extraction/Reduction - This tool instruments operational programs and allows information on dynamic performance and accuracy to be extracted. The reduced data simplifies analysis of bottlenecks, and helps to identify poorly-performing programs.

System Builder - This tool supports the SYSGEN process. It collects all of the programs, data and system instructions needed to create an operational system. It might perform consistency or completeness tests, schedule recompilation of source text, and automatically create jobs for the linker/loader. The product is a generated system for ECS.

Ada Run-Time Support - This tool would provide the run-time functions to support compiled Ada programs.

2.4.4 Requirements on the SEE

Implementation will place the following requirements on the SEE:

Baselined Products - The baselined products are the source text, object modules, load modules, and any required system generation information. The latter might include a job control language sequence to create the system plus the necessary system configuration information and data.

Non-Baselined Data - This information includes work-in-progress, test data, drivers and stubs, and debug and other analysis information produced by development tools.

Measurements - Examples of useful measurements include data on the products of programming activities, data on resources expended, data on changes and errors, and subjective measurements of the quality of the implementation.

2.5 Correctness Analysis

Correctness analysis shows that the implementing system satisfies the specified purpose of the system. Since the purpose is first formally captured in the specification, correctness analysis begins by showing that the specification reflects the requirements.

Figure 2.5-1 provides an overview of the scope of correctness analysis (Ferrentino and Mills [77]). Many software developers are concerned with testing the completed system against the requirements without regard for the adequacy of the requirements. This approach to correctness analysis is unsatisfactory for the following reasons:

- The requirements may be inconsistent or incomplete for proper capture of the intended purpose of the system.
- Finding and correcting errors after system implementation is much more costly than correcting errors where they are introduced.
- Correctness analysis improves the quality of software. Quality cannot be tested into a system.

Thus, correctness analysis, as shown in Figure 2.5.1-1, consists of continuous analysis throughout the software engineering process as well as testing of the implemented system.

2.5.1 Characteristics

The main focus of correctness analysis is not reduction in the number of errors introduced into the system, but reduction in the error-day measure. The error-day measure is the average length of time (in days) that an error goes undetected. A system with a short error-day measure is likely to be of high quality and low development cost because of reduced reliance on testing to remove errors. If a system has a long error-day measure, the system is likely to be costly to develop and unreliable.

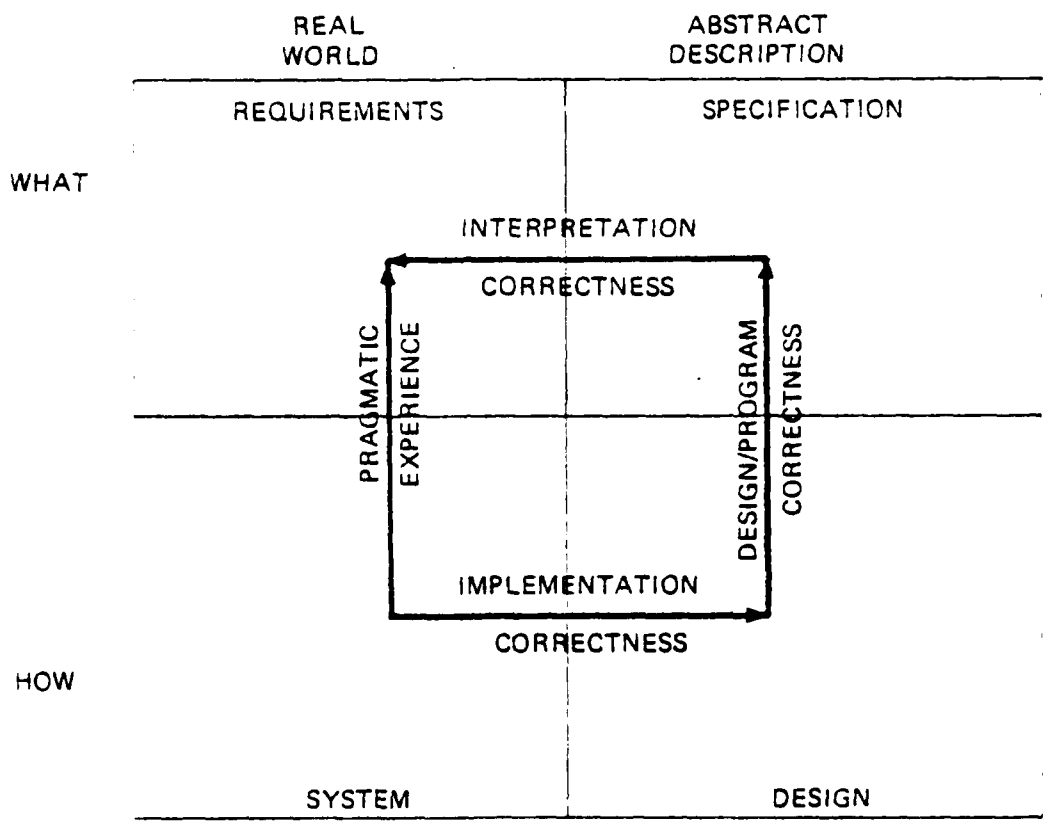


Figure 2.5-1: The Scope of Correctness Analysis

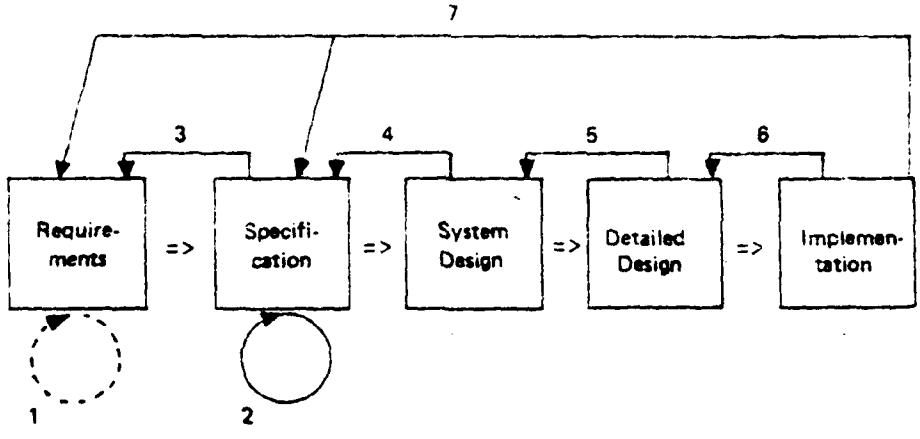


Figure 2.5.1-1: Span of Correctness Analysis

Correctness analysis must span all activities to minimize the error-day measure of a system. Figure 2.5.1-1 depicts this coverage. The arrows indicate correctness analysis activities as described below:

- Review of the requirements for consistency and correctness. Because requirements are rarely defined rigorously, the arrow is dotted. (1 on the figure.)

- If completeness and consistency cannot be established at the requirements level, completeness and consistency should be established with the specification as a base. (2)

- Determining that the specification correctly implements the requirements. (3)

- Determining that the system design correctly implements the specification. (4)

- Determining that the design decomposition does not violate the system design. (5)

- Determining that each unit of the implemented system correctly reflects the detailed design and that the implemented system correctly represents requirements and specifications. (6 and 7)

Correctness analysis must be an integral part of each phase of an incremental development plan. If incremental development persists throughout the life cycle, these activities apply to maintenance. When changes are made, a full analysis of the system is not needed as only the changed areas and the areas affected by the changes need to be analyzed.

2.5.2 Methodologies

The methodologies linked with correctness analysis are either static analysis or dynamic analysis. Static analysis includes, in order of increasing rigor, reviews, inspections, and proofs of correctness. Dynamic analysis includes all testing techniques.

Reviews - Reviews determine the internal completeness and consistency of system requirements and software specification, design and test information. They also assess its consistency with its predecessor information. Reviews involve a broad range of people, including developers, managers, users, and outside experts or specialists. A review must have specific objectives and questions to be addressed (Heninger [78]). The review findings generate rework tasks for the development group.

Inspections - Inspections evaluate the correctness of component level specification, design, code, test plans, and test results. They are more formal and rigorous than reviews. An inspection involves a small group of people of a specific make-up, and follows a well-defined procedure (Fagan [75]).

Proofs of Correctness - All development products should be verified with an informal proof of correctness. Certain critical kernals of code or special applications may require a formal proof of correctness.

Testing - Dynamic execution of the system or system component with known inputs in a known environment is a "test." If the test result is consistent with the expected result, the component is deemed correct in the limited context of the test. The following baselined documents are created relative to testing:

- Test Plan - Defines the scope, approach, and resource needed for testing.
- Test Procedures - Provides a detailed description of the steps and test data associated with each test case.
- Test Results - Documents the results of each test run. Unsuccessful runs trigger trouble reports which must be addressed by the development group.

There are two approaches to testing -- black-box testing and white-box testing. Black-box testing uses only knowledge of externals (to the function) while white-box testing uses knowledge of the internal design of the function.

Black-box testing uses the specification to develop test cases (Howden [76]) and is most appropriate for system testing because it directly demonstrates that the implemented system satisfies the specification. White-box testing uses design information to develop test cases (Miller and Melton [75]) and is most appropriate for component testing.

The relationships between system functions and component or system test cases should be clearly established. Then, when changes are made to parts of a system, a subset of test cases can be identified which will test the system sufficiently. This process is called regression testing. Effective regression testing is a good way to reduce software development costs.

2.5.3 Supporting Tools

A representative set of tools to support the correctness analysis methodologies is listed in Figure 2.5.3-1. Examples of all these tools may exist today but not all of them could be used. In many categories, they are not effective or they were implemented in a narrow environment such as for FORTRAN only. A brief description of each tool is given below.

Performance Model - Performance models evaluate system performance constraints such as response time. The evaluation is performed by an analytic model or simulation model based on the system design. If the performance model is used to evaluate requirements feasibility, a high-level design is assumed. Performance models have been developed for many systems. The Navy is currently developing a generalized modeling tool called Performance Oriented Design.

Prototyping Aid - Developing an operational prototype will allow evaluation of requirements and design approaches. Executable design languages might be useful in this regard; there are several examples of these, such as APLGOL used on the LAMPS program.

Consistency/Completeness Analyzer - These tools aid analysis of internal consistency and completeness when specification requirements are expressed in a

Methodology	Tool
Review	Performance Model Prototyping Aid Consistency/Completeness Analyzer
Inspection	Standards Checker Assertion Analyzer
Proofs of Correctness	Symbolic Execution Theorem Prover
Component Test	Test Harness Test Data Generator Hardware Simulator Host-Targeted Compiler Test Results Comparator
Sytem Test	Environment Simulator/ Stimulator Performance Monitor Data Extraction and Reduction Scenario Generator Black-Box Test Generator
General	Reliability Model

Figure 2.5.3-1: Methodologies and Tools of Correctness Analysis

machine-interpretable form.

Standards Checker - Standards checkers perform static analysis of code or documentation and identify standards violations. Some tools address part of this job but none known to the authors would conform to Navy MIL-STD-1679.

Assertion Analyzer - These analyzers verify that code correctly implements specification by checking the truth of the assertions (embedded in the code) against actual program execution.

Symbolic Execution - Symbolic execution is useful for proving the correctness of certain classes of programs. It involves "executing" them symbolically to prove certain assertions. The existing tools are aimed primarily at FORTRAN programs.

Theorem Prover - Theorem provers automate proof-of-correctness techniques. This technology is not in use by production software groups.

Test Harness - A test harness provides the framework for unit testing of procedures. With it, programmers can interactively define the procedure interface, prepare test data, run an instrumented test, and display the test result.

Test Data Generator - Based on a given testing strategy such as "test every path," this tool will automatically develop test cases based on the code (or perhaps the design).

Hardware Simulator - Hardware simulators allow object code for a target computer to be tested on the host computer.

Host-Target Compiler - A software alternative to hardware simulators which use source text to generate object code that executes on the host computer.

Test Results Comparator - This tool compares actual unit test results against expected results. It issues a trouble report if the test results are not correct.

Environment Simulator/Stimulator - The simulator

duplicates the operational environment of the ECS in a test facility. This is done before integration with the sensor and weapon systems. The stimulator tests the system by simulating input such as sensor and weapon interface signals.

Performance Monitor - These tools permit measurement of system performance parameters, such as response time, algorithm processing time, channel utilization, etc.

Data Extraction and Reduction - These tools help to analyze the system dynamically. During execution, data is captured and stored, and later reduced by processing to create reports for the dynamic analysis activity.

Scenario Generator - These tools control the complex parameters which create the scenarios and drive the simulation of the environment.

Black-Box Test Generator - These tools generate functional test skeletons by using the specification of the software system. The analyst then completes the scenarios by adding detail to the skeleton. The tools should include sampling techniques to assure adequate coverage of the specification.

Reliability Model - These tools use the error history of the system over its life cycle to estimate a reliability measure for the system.

2.5.4 Requirements on the SEE

The requirements on the SEE data base, derived from the correctness analysis, are summarized below.

Baselined Products - Test plans, test procedures and test results (of correctly executed tests) are all baselined. They are controlled by configuration management. The results of inspections and proofs might also be baselined.

Non-Baselined Data - The non-baselined data includes work-in-progress, static analysis data, trouble reports, and debug data. Temporary storage of this

type of information is required.

Measurement Data - A number of measurements associated with correctness anlaysis should be captured. These include: number of modifications to a unit, number of errors found per unit, number of test runs, number of errors by error category, and test coverage.

The support system must provide a framework for each tool identified in Section 2.5.3. In particular, the demands on the support system levied by system test can be extensive. One or more dedicated computers may be required to host the environment simulator to support a system test of an ECS. Special hardware may also be required to interface the environment simulation computer with the target computer or subsets of weapon systems.

2.6 Management

Management has the responsibility of ensuring that the software meets its delivery goals of schedule, cost and quality. Figure 2.6-1 illustrates the management process.

Planning and organizing resources results in the creation of a development plan which shows how the resources, applied within the software development process, will meet the delivery goals of schedule, cost and quality. The plan must also be able to accomodate modifications due to authorized changes and unexpected problems. Monitoring evaluates progress according to plan, effectiveness of development activities, and the quality of the products (intermediate and final). The control activity is used to take corrective action when necessary.

2.6.1 Characteristics

Planning: The activity of planning involves estimating what is needed to meet the objectives of the project, and then organizing and allocating the resource to perform the work. Estimation must consider the size and complexity of the software being developed, the skill mix and productivity of the development professionals, the schedule, potential problem areas, and the power and sophistication of the sup-

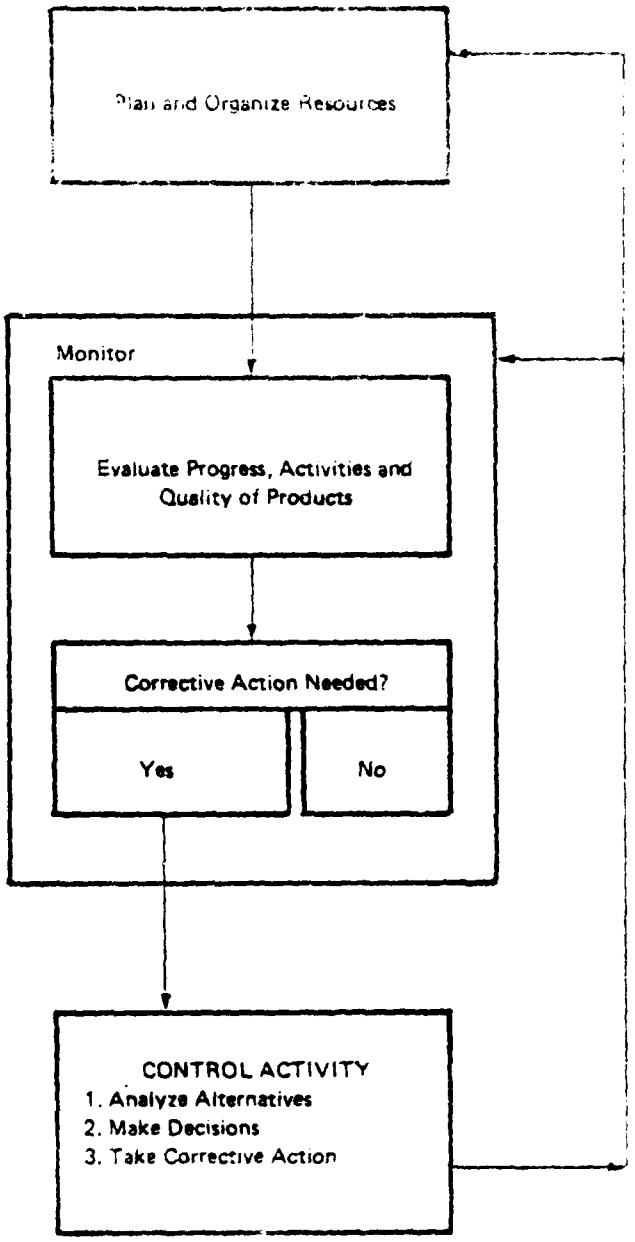


Figure 2.6-1 The Management Process

port resources. Once estimation is complete, the resources will be organized and allocated to do the work.

Monitoring: This management activity is used to guage the progress of the work to date, the quality of the products to date, the quality of the development process, and the adherence to standards. Management must be constantly aware of the expenditure of resource to meet the project goals and the progress of the work toward the milestone dates. Management must take corrective action on expenditures or progress which is not according to plan.

Using quality assurance techniques, the quality of each product is compared to a pre-established index of quality. Products not meeting the standards would require rework. Accomplishment against plan is checked when progress is evaluated. Technical progress and resources expended must be matched against the budgets and schedules of the development plan, and deviations marked for correction.

Quality assurance techniques are also used to check the quality of the development process and adherence to standards. The techniques used are similar to those of correctness analysis. Reviews, inspections and test results are audited and analyzed with deviations noted for corrective action.

Controlling: Control must be maintained over both the development process and the products of the process. The control of the process focuses on meeting the goals and objectives (established in the development plan) through the use of available resources. Authority is delegated and responsiblity assigned through an organizational structure.

The control of products focuses on the products of each activity in the software development process. Configuration management protects products by creating a master copy (i.e., a "baseline") against which controlled changes are made. This ensures the consistency of the evolving system as requirements are defined, specifications are documented, the design is developed and implemented, and changes made.

[illegible] 2 Methodologies

[illegible]

Estimation - Most resource estimation methodologies use the measurements from prior projects to estimate resources. Examples are found in Bailey and Basili [80], and Putnam [73]. Support of estimation methodologies requires a data base of comprehensive measurements including such software system parameters as size of source code, source language, development resources expended, and complexity measures.

Precedence Networks - This planning methodology is used to analyze task dependencies and to determine the critical path of development activities. Such an analysis is usually needed to define a realistic schedule. It is also useful in evaluating contingencies and creating contingency plans.

Change Control - This is the core of configuration management. It controls all changes to baselined products. The approval process for changes might be as follows.

- The written request for change is submitted to the configuration management function. It might come from a change in requirements or from a trouble report documenting a defect.
- An assessment is made of the technical feasibility of the change, and its impact on schedule and budget.
- The change is approved or disapproved based on its value and cost.
- The development plan is modified and resources adjusted to add approved changes.
- The fully verified change is entered into the new baseline.

2.6.3 Supporting Tools

Many tools are applicable to management and can support the methodologies and activities described above. Some of the more useful tools for management include:

Resource Estimation Model - This software uses a data base of measurements on past projects, along with a description of the new system, to create resource estimates for development.

Automated Precedence Network - This software creates precedence network charts and determines the critical path based on the input of detailed milestones and precedence relations.

Automated WBS - This tool helps to create budgets and a work breakdown structure.

Schedule Generator - This tool uses output from the precedence network, and organizational responsibilities (related to the WBS), to create schedules by organizational entity.

Change Request Tracker - This tool logs change requests when submitted, tracks them through the approval cycle, and records their resolution.

Resource Scheduling Aids - These tools permit resources, such as computers, conference rooms, terminals, and test equipment, to be scheduled. Usage reports on the resources and the scheduling of the resources are the main functions of these tools.

Report Generators - Report generators create management reports on technical, budget, and administrative status.

2.6.4 Requirements on the SEE

The activity of management imposes the following requirements on the SEE data base.

plan, configuration management data and quality assurance plans should also be baselined.

Non-Baselined Data - Significant amounts of information associated with the management must be kept temporarily. This information includes engineering change requests, trouble reports, resource allocation plans, actual resource utilization reports, technical milestone status, action item status, and the results of quality assurance reviews.

Measurement Data - Many measurements are of interest to management. These include the number of engineering change proposals (ECP), and trouble reports (TR), time to process an ECP or TR, resource use for each ECP or TR, resource use by project activity, and software size and complexity measures.

The support system must have certain minimum capabilities to support management. These capabilities go beyond providing for the tools identified in Section 2.6.3. The support system must provide an authorization mechanism that allows management to protect the project data base from outside influences. It should also limit access by internal users. On the other hand, management must be able to monitor all aspects of the project including the evolving products.

3. An Abbreviated Software Engineering Process

The software engineering process described in Sections 1 and 2, if rigorously applied, could significantly improve development productivity and software reliability. Optimistically, it might allow production groups to produce two or three times as much as they do now. Unfortunately, this level of improvement will not be enough to meet the Navy's projected demands over the next ten years. An order of magnitude improvement in productivity (or more), is needed. Thus, methods for shortening the software engineering process are needed.

Thus far, the main focus of automation has been the mechanical or clerical aspects of the software engineering process. Actually, this semi-automates only a small part of the entire process. The quickest way to abbreviate the process of developing software is to learn how to adapt existing software to new software systems. This is analogous to prefabricated parts, the technique which revolutionized the industrial process. Some, or even all, of the design, implementation and correctness analysis activities can be eliminated for parts of a software system under development.

The potential for savings is significant. Although most software development today is treated like the first of its kind, there are few instances where similar functions have not been implemented already. Even in an ECS, where the complex nature of the system suggests that each new system is unique, there is much common function from system to system. For instance, command and control systems are largely a collection of functions common across different C^2 systems, e.g., DBMS, display, message processing, communications, etc. If we capitalized on this commonality in the context of a modern software engineering process, order of magnitude productivity improvements are feasible.

Two concepts which provide a basis for reusable software are common components and application skeletons. Common components are programs which embody a certain function or class of functions which might be useful in several systems. (An example is a data base management system.) An application skeleton is a partly implemented software system designed for a class of application. It can be tailored to a specific application in the class. An example is a sim/stim system skeleton comprising simulation problem control, physical models, and a man/machine interface for simulation control. However, it requires developing specific weapons simulation code and interfaces to the target ECS of concern. The sim/stim application skeleton might represent 70% of the completed system.

For common components and application skeletons to become a real force in productivity improvement, several conditions must be met. Just as industry had to have standards for the use of prefabricated parts, common components and application skeletons will also require rigorous standards. These standards will have to address interface and development guidelines. Once standards are estab-

lished, a continuing investment will be required to develop a full library of components and skeletons. Tools to [illegible] emerging technology of knowledge-based expert systems. Other tools to aid efficient tailoring of common components or application skeletons also will be needed.

The software engineering process described in Sections 1 and 2 provides contextual requirements for a SEE. The full support of common components and application skeletons also is a requirement on the Navy SEE. This support can help the Navy meet its projected needs for software.

PART II

DESCRIPTION OF A SOFTWARE ENGINEERING ENVIRONMENT

Part II: Description of a Software Engineering Environment

In Part I, a software engineering environment (SEE) was characterized as an integrated system consisting of a computer-based support system and data base. The purpose of the SEE is to support all personnel involved in the development and continuing adaptation of software, according to a well-defined doctrine of software engineering disciplines. Part I outlined such a doctrine to provide the context for the SEE requirements. Part II provides requirements for the SEE.

Although the approach taken in this requirements document assumes a close tie of the SEE requirements to a unified set of software engineering disciplines, not all efforts to build environments have followed this line. One alternative is the so-called "tool kit" approach wherein a set of tools (possibly integrated) is provided, with no implied constraints on the choice of methodologies or use of the tools. An example of a tool kit approach is the Unix Programmer's Workbench (Dolotta [76]). The tool kit approach is viable for some environments supporting code generation. When requirements analysis, specification, and design are to be included, the tool kit approach is inadequate as these activities involve a high degree of intellectual and creative activity.

The primary characteristic of the SEE, as defined in this document, is that it is methodology-driven. However, several requirements are included which are derived from useful tool kit concepts. For instance, tools to build tools have been proven valuable in tool kit approaches and are included in the SEE requirements.

The SEE must satisfy several general requirements. In summary, they are:

- Scope - The SEE must support all activities of the life cycle of software as defined in Section I.1.2.

- Application - The SEE must adequately support the development and continuing adaptation of Navy ECS.

- Rehostable - The support system and its data base

must be able to be rehosted on commercial and military computers.

- Evolution - The SEE must be flexibly designed as software engineering methodologies and tools are still evolving.

- Re-use - The SEE must encourage the capture of proven designs and implementations for re-use without re-invention

- Reliability/Availability - The support system must be reliable and stable enough to support long periods of uninterrupted operation.

The SEE requirements are described in the following subsections. Section 1 addresses the SEE data base. It describes the content and different views of data that must be supported. Section 2 addresses the requirements for the support system. These requirements are described in terms of characteristics of external interfaces, functional capability, performance constraints, and design constraints.

1. Data Base

The data base of the SEE is its major integrating element. It will contain the current state of the software system, the current state of the development process, and historical data reflecting previous states of development. In short, it will retain all the information on the activities of the various personnel working on a project.

The clear implication is that the SEE data base will contain far more data than currently stored in library systems and specialized files today. For this reason, the design of the data base and its relation to the support system components is a key concern in the implementation of a modern SEE.

This section of the document presents requirements for the SEE data base. An effort has been made to avoid any design implications which might overly constrain future SEE designs. Therefore, the contents of the data base and relations among data objects in the contents, are described but no logical or physical structure for the data is imposed.

The contents of the SEE data base have been put into categories, rather than simply listed, to aid in understanding the data base. Many choices for categories were possible. The categories finally chosen represent classes of data having common control attributes. As an example, the "baselined" category contains all data objects which will be controlled by configuration management. The five major categories chosen for data objects are:

- Baselined - Any data object considered to be a part of the software system baseline and thus under control of configuration management.

- Non-baselined - Any information which is transient or represents "work-in-progress" towards a baselined object.

- Measurement - Metrics derived from the development process activities for use by management, software analysis, or historical quantitative analysis.

- Archive - Any quantitative or qualitative data from

any of the other four categories retained for historical purposes.

- Support System Library - All of the software and data associated with the support system including tutorial data, standard error messages, and SYSGEN data.

Besides the view of the data base defined by the above categories, macro relations and micro relations must also be supported. These views are defined in the relations section of the data base description as macro relations, (those relations that encompass a large number of data base objects). There are also many micro relations (those affecting small numbers of objects) that must be defined in the data base design; these are also described.

The following sections on the data base contents and relations capture the essentials of the SEE data base. However, many details have been omitted due to the magnitude of completely defining all data and data relations. It is felt that the degree of detail is sufficient for the intended purpose of this document.

1.1 Contents

The contents of the SEE data base are described below in the five categories outlined in Section 1. The contents are described in terms of coarse objects, e.g., design information, test plans, and temporary source areas. A standard format is used for all data object descriptions to permit conciseness and ease of reference; the format is illustrated below.

OBJECT NAME (*): description of the data object.

[List of undefined data items or elements composing the data object.]

Forms: [text or tables or graphics or language or encoded.]

Reference: [reference to discussion relating to data object in Part I or II.]

Notes:

1. The asterisk (*) will be either (R) for required or (O) for optional; required items are those for a mandatory for inclusion in a SEE while optional ones are not.

2. The term "text" under Form(s) means English prose.

3. The term "language" covers all formal languages used in a SEE: a formal grammar structure, a formal semantics language, formal mathematical expressions, JCL (job control language) procedures, or formal programming languages.

4. The term "encoded" refers to computer-created executable code which is either translated from source text or combined with other executable code modules to create executable load modules.

1.1.1. Baselined Products

A baselined product is any data object which is "owned" by management and controlled by the configuration management process. This implies that baselined products persist over the life of the software system and that changes to these objects are controlled very closely. The SEE must provide a means to manage change deltas to baselined products such that an audit trail is possible.

In development projects today, the baselined products are the source and object code, generated systems, and supporting documentation. Some of these baselined products are in electronic form, while others may be maintained as printed documents. In the SEE, all baselined products will be maintained in electronic form in the SEE data base. Printed documents can be created at any time using the information stored in the SEE data base.

The baselined products are described below, organized by the development process activity which creates them.

A. Requirements Analysis

SEMANTICS INFORMATION (R): The captured semantics of the requirements for the software system.

[Definition of terms, relations among terms, references to documents containing the requirement, references to other baselined products for traceability]

Form(s): text, language, graphics

Reference: I.2.1

B. Specification

SOFTWARE SYSTEM SPECIFICATION (R): The description of what the software system is to do; the specification must be written in a formal notation.

[External interfaces, operational modes, output functions]

Form(s): text, tables, language, graphics

Reference: I.2.2

C. Design

SOFTWARE DESIGN (R): The description of how the system is to be implemented; the description must be written in a formal notation.

[Module hierarchy, module specifications, abstract data structures, process design language (PDL)]

Form(s): text, language, graphics

Reference: I.2.3

D. Implementation

SOURCE CODE (R): The source code for all programs; the code is written in the appropriate programming language.

Form(s): language

Reference: I.2.4

OBJECT CODE (O): Executable code translated from the baselined source code.

[Executable code, external references, static analysis data.]

Form(s): encoded

Reference: I.2.4

PATCHES TO OBJECT CODE (O): Executable patch[1] code either written in absolute machine code or translated from baselined source code.

[executable code, external references, static analysis data; identifier for source code being corrected, error identifier, and appropriate version numbers]

Form(s): encoded

Reference: I.2.4

---

1 Occasionally software errors in an ECS will require immediate fixes, and the correction cannot be handled quickly through the normal error correction cycle. Such an immediate fix is called a "patch." Patching is notorious for its introduction of additional errors because the patch, as a quick solution, was inadequately analyzed and researched and tested (subtle interaction errors are especially prevalent). Patching can also be hazardous if it is not carefully controlled for inclusion into the source [illegible] Navy ECS software has requirements for post-[illegible] errors, the software engineering methodologies and procedures for [illegible] object code must recognize the need for a patch capability. It is the responsibility of [illegible] patching as a tool to test and debug software [illegible]

****

SYSGEN DATA (R): Any data required by the system generation tools to create system load modules including parameters for operating systems, executives, or configuration definition.

Form(s): table, encoded, language

Reference: I.2.4

SYSTEM LOAD MODULE (O): An image of the software system in machine-executable form.

Form(s): encoded

Reference: I.2.4

E. Correctness Analysis

CORRECTNESS ANALYSIS PLAN (R): A description of the total correctness analysis approach to be taken for requirements analysis, specification, design and implementation; the plan will also include resource requirements and schedules.

Form(s): text

Reference: I.2.5

TEST PLAN (R): A description of the approach to be taken specifically for system test; it will include resource requirements and schedules.

Form(s): text

Reference: I.2.5

TEST PROCEDURES (R): Description of the individual tests making up system test and unit tests.

[illegible] procedure.

[illegible] text, language, tables

Reference: I.2.5

TEST RESULTS (R): The test results in [illegible] form.

Form(s): text, encoded

Reference: I.2.5

PROOF OF CORRECTNESS (O): Formal proof of correctness of a design or an implemented procedure.

Form(s): text, encoded

Reference: I.2.5

F. Management

DEVELOPMENT PLAN (R): The current plan governing all development or continuing adaptation activities.

[Organization, work breakdown structure, resource estimates, budgets, progress status, and schedules]

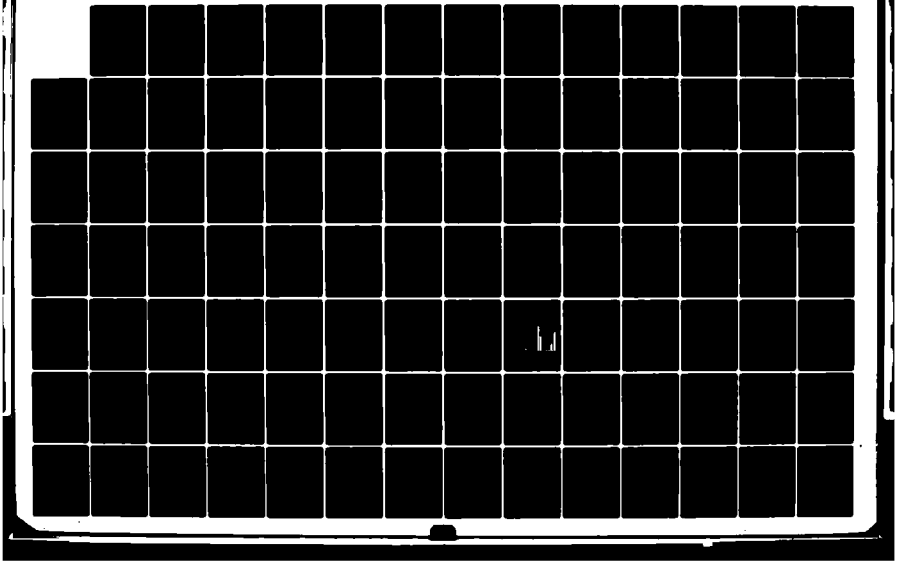
Form(s): text, tables, graphics

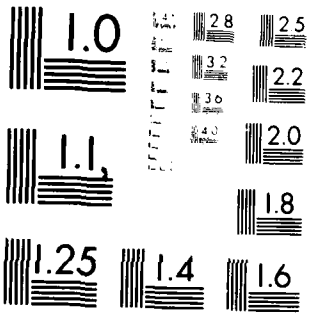
Reference: I.2.6

STANDARDS (O): The current Navy or project standards for requirements, specifications, design, implementation (program code and test), correctness analysis of all phases, quality assurance (QA), configuration management (CM), and management.

AD-A131 941 A SOFTWARE ENGINEERING ENVIRONMENT FOR THE NAVY(U) 2/3
NAVAL MATERIAL COMMAND WASHINGTON DC 31 MAR 82

UNCLASSIFIED F/G 9/2 NL





Form(s): text, tables

Reference: I.2.6

ECP/TR LOG (O): A log of the status of all engineering change proposals or trouble reports.

[Date of acceptance, organizational entity assigned, date completed coding, etc.]

Form(s): text

Reference: I.2.6

G. General

DERIVATIVE DOCUMENTS (O): Data objects used to generate any standard documentation, e.g., a User's Manual which might be derived from information in the specification or design data objects.

[User's manuals, systems programmer manuals, SECNAVINST 3560.1 documents, etc.]

Form(s): text, tables, graphics, language

DOCUMENT TEMPLATES (O): Shell of standard documents that can be filled from other baselined data objects to create derivative documents.

Form(s): text, encoded

AUXILIARY DOCUMENTS (O): Documents which are not derived from baselined data objects and which may be defined as baselined data objects, e.g., externally-created documents entered via the SEE text editor.

Form(s): text, tables, graphics, language

1.1.2 Non-Baselined Data

The salient characteristic of non-baselined data is that it is temporary in nature (i.e., working material), and is still in the iterative process. People with assigned work must have space to work. These spaces are used to store work-in-process, results of analysis runs, schedules, etc. Devices such as "mailboxes" (for message communication among project members) would also be included in this category.

An individual space should be defined such that data objects within it are "owned" by the individual and access by others is controlled by the individual. This space access must not be so restrictive as to limit management review of technical status.

Examples of data objects which might reside in work-spaces are described below by each development process activity.

A. Requirements Analysis

TRIAL DESIGN (O): Preliminary designs of sufficient detail to support analysis of the feasibility of requirements.

Form(s): text, language, graphics

Reference: I.2.1

MODEL INPUT (O): Model input parameters describing a trial design.

Form(s): text, language

Reference: I.2.1

MODEL RESULTS (O): Results of a modeling run evaluating a trial design.

Form(s): text, table

Reference: I.2.1

PROTOTYPE (O): An operational prototype of a software system.

Form(s): language

Reference: I.2.1

SEMANTICS ERROR LIST (O): A list of incomplete or inconsistent areas identified in the semantics information by a semantics analysis tool.

Form(s): text

Reference: I.2.1

B. Specification

PARTIAL SPECIFICATION (O): A specification, or modifications, leading to a baselined specification.

Form(s): text, tables, language, graphics

Reference: I.2.2

SPECIFICATION ERROR LIST (O): A list of incomplete, inconsistent, or syntactically incorrect information comprising the specification.

Form(s): text, table

Reference: I.2.2

C. Design

PARTIAL DESIGN (O): An incomplete software design, or design modifications, leading to a baselined software design.

Form(s): text, language, graphics

Reference: I.2.3

DATA FLOW DIAGRAMS (O): A graph diagramming the various states and transformation of data as it passes from input to output.

Form(s): text, graphics

Reference: I.2.3

HIERARCHY CHARTS (O): A directed graph depicting a hierarchic relation among software components.

Form(s): text, graphics

Reference: I.2.3

D. Implementation

PARTIAL PROGRAMS (O): Prior to baselining: An untested program, a partially developed program, or a modification for a baselined program.

Form(s): language, encoded

Reference: I.2.4

STUBS (O): Stubs used in unit testing of a program.

Form(s): language

Reference: I.2.4, I.2.5

STATIC ANALYSIS DATA (O): Output of a static analyzer run against a partial program.

Form(s): text, tables

Reference: I.2.5

DEBUG DATA (O): Output of a debug run against a partial program.

Form(s): text, encoded

Reference: I.2.4

E. Correctness Analysis

PARTIAL TEST PROCEDURES (O): Test procedures in development prior to baselining.

Form(s): text

Reference: I.2.5

INVALID TEST RESULTS (O): Results of test runs against a partial program.

Form(s): text, encoded

Reference: I.2.5

TROUBLE REPORTS (O): A report of an error detected in the baselined software system.

Form(s): text

Reference: I.2.5, I.2.6

F. Management

COMPUTER SCHEDULE (O): Current schedule for computer use by various organizational units.

Form(s): text

Reference: I.2.6

BUDGET TO ACTUALS (O): An up-to-date comparison of actual resource used against budgeted resources, including manpower, time, and computer resources.

Form(s): text, tables, graphics

Reference: I.2.6.

MILESTONES (O): An up-to-date record of status against milestones as defined in the baselined development plan.

Form(s): text, table, graphics

Reference: I.2.6

ALTERNATIVE PLAN ESTIMATES (O): Resource estimations associated with alternate plans for recovering from unexpected delays or resource expenditures.

Form(s): text

Reference: I.2.6

G. General

BULLETIN/MAIL BOX (O): A mechanism for broadcasting news of system or project interest, or for sending a message from one individual to another; (optionally) displayed on the interactive text device or included on any user-generated output.

Form(s): text

****

1.1.3 Measurement Data

Measurements are data objects of a quantitative nature representing some characteristic of the development process or the software product. This type of data might be used by the technical personnel to evaluate the software product in development, or it might be used by management to assess the quality of the product or the health of the project; or it might be used by analysts to understand how the development process could be improved. The measurement data subjects two subcategories - product and process. Product measurements are any quantitative data characterizing the software system itself, and might include complexity and reliability measures. Process measurements are any quantitative data characterizing the development process.

Measurement data is owned and controlled by managers and only management can change or discard it. Read access to measurement data should be unrestricted.

Measurement data should be captured automatically. This requires mechanisms built into the support system. It also implies that measurements will be captured concerning operations on an individual's work space, which allows management to view work in progress.

The measurement objects described below in the product and process subcategories are static in nature as they represent a quantitative measurement at some point in time. For the measurements to have meaning analytically, it must be possible to determine the time derivative (i.e., rate of change) of many of the objects. This means the SEE must time-tag measurements before they are stored. The SEE data base must be able to store sequences of time-tagged measurements representing an object type.

A. Product Measurements

Examples of measurement data in the product subcategory are given below.

NUMBER OF SEMANTIC ENTITIES (O): A count of the semantic entities in the semantics information.

Reference: I.2.1

****

NUMBER OF SEMANTIC RELATIONS (O): A count of the one-to-one relationships among terms in the semantics information. The number of semantic entities and the number of semantic relations provide a measure of complexity.

Reference: I.2.1

NUMBER OF SPECIFICATION FUNCTIONS (O): A count of the distinct functions identified in the specification.

NUMBER OF MISSING REQUIREMENTS (O): A count of the specification inputs, outputs, operational modes, functions, or design constraints for which no traceable requirement has been identified through the semantics information.

Reference: I.2.1, I.2.2

NUMBER OF MODULES (O): A count of the modules defined in the system level design.

Reference: I.2.3

NUMBER OF PROGRAMS (O): A count of the programs in the detail level design.

Reference: I.2.3

NUMBER OF LINES OF SOURCE TEXT (O): A count of the lines of non-commentary source text related to the operational software system.

Reference: I.2.4

TYPE OF SYSTEM (O): A count of the total number of [illegible] code and data in the operational system.

Reference: I.2.4

RESIDENT MEMORY (O): A count of the number of bytes of memory required by the nucleus and transient areas of the operational system.

Reference: I.2.4

NUMBER OF KNOWN DISCREPANCIES (O): A count of the known discrepancies between the specification and the requirements, or between the implemented software and the specification.

Reference: I.2

RELIABILITY (O): A statistical estimate of software component or system reliability based on error history and (perhaps) complexity measures.

Reference: I.2.5

B. Process Measurements

NUMBER OF ERRORS FOUND (O): A count of the number of errors found in the software products.

[by product (specification, design, etc.); by increment, release or version; by types of error; by severity (i.e., effort needed to fix)]

Reference: I.2

MAN MONTHS EXPENDED (O): A count of the man-months of effort expended.

[by organizational entity; by life cycle activity; by product; by increment, release, or version; by individual]

Reference: I.2.6

INDIVIDUAL PRODUCTIVITY (O): A count of the man-months of effort expended in certain activities.

[by product, by version]

Reference: I.2.6

NUMBER OF CHANGES (O): A count of the number of changes made to a baselined product.

[by release, by version, by life cycle activity within increment]

NUMBER OF COMPILATIONS (O): A count of the number of compilations of a program before acceptance for baselining.

NUMBER OF TESTS (O): A count of the number of unit tests of a program before success.

1.1.4 Archive Data

Archive data will contain a history of all projects including the project's data base and especially baselined data. Archive data should be kept after the development phase and probably after the operational life of the system. Analysis of it will help software professionals learn from the past, both from successes and from failures. It should be mandatory for the baselined data (of the developed system) to be available, either online or in archive form, during life cycle support. Archive data, because of its potential volume and infrequent access, will be stored off-line. Media such as magnetic tape or microfilm would be appropriate.

Archive data might include data objects from any of the other four categories. For this reason, no data objects are explicitly defined for this category.

One needs to be able to find and retrieve the desired data objects to make archived data useful. An index of

archived data must be created and maintained. Indexing should be based on the time of archival, the project name, the software system name (as modified by release and version); all of these indices are needed. The index must contain a brief description of the data object and pointers to where the data is stored.

1.1.5 Support System Library

A support system is an integrated set of computer-bas tools supporting all of the activities of the software eng neering process. The support system library will contain all of the tools comprising the support system; it will also contain data assumed to be part of the support system such as tutorial information, error information or SYSGEN data. The support system library is considered baseline data; therefore, the library and all modifications to it are under configuration management.

The data objects of the support system library category are described below.

SYSTEM SOFTWARE (R): Systems software components including operating systems, data base management system, telecommunications handlers, utilities, etc.

[specifications, design, source code, object code, etc.]

Form(s): text, language, encoded

Reference: II.2.2.1

GENERAL SERVICES (R): Software components of the support system for general application use such as a text editor and document generator.

[specifications, design, source code, object code, etc.]

Form(s): text, language, encoded

Reference: II.2.2.2.1

ACTIVITY-SPECIFIC TOOLS (R): Any software tool that supports a methodology used in the life cycle activities.

[specification, design, source code, object code, etc.]

Form(s): text, language, encoded

Reference: II.2.2.2.2

USER-GENERATED TOOLS (O): Any tool created by an individual or project for limited application.

[specification, design, source code, object code, etc.]

Form(s): text, language, encoded

APPLICATION PARADIGMS (O): Reuseable components or program skeletons used to reduce development time and cost.

[specification, design, etc.]

Form(s): text, language, encoded

Reference: II.2.2.2.3

TUTORIALS (R): Information provided when "help" is requested by an interactive user.

Form(s): text

1.2 Relations Among Data Base Objects

One of the unique aspects of the SEE data base is the extensive degree of interrelation among the data objects described in Section II.1.1. These interrelations will greatly complicate the job of designing the data base structure and data base management system (DBMS). The

purpose of this section is to describe the relations that must be accommodated in the SEE data base.

Not only must one-to-one and one-to-many relations be accommodated, as in a hierarchically structured data base, but many-to-many relations must be accommodated as well. For example, the "linked from" relationship is many-to-many because one object module can be linked from many other object modules, and many object modules can be linked from a single object module. If the number of many-to-many relations in the data base is high, a relational DBMS for the SEE might be more appropriate than a hierarchical DBMS. The issue of hierarchical versus relational organization for the data base needs further investigation. Analysis is needed of such key factors as how the size of the data base and the activity volumes impact performance.

Two classifications are used to describe the SEE data base: macro relations and micro relations. Macro relations tie many objects of the data base together to form a "meta object," an object with many parts which need to be referenced as whole. Macro relation data will allow a meta object to be addressed with one reference. An example of a meta object would be a list of all baselined source procedures belonging to Project A. Micro relations reference two or more objects in the data base.

1.2.1 Macro Relations

There are nine macro relations. Five are categories described in Section II.1.1: baseline data (and products), non-baseline data, measurement data, archive data, and the support system library. The other four macro relations are: project, release, version, and increment. Since the first five categories have been described, only the last four are defined below.

> Project - A project is an organizational entity with an assigned development responsibility. Therefore, a project macro relation identifies all the SEE data objects belonging to (controlled by) the project. These might include data objects in the baseline, non-baseline, measurements, archive, and support system categories.

Release - A release is a specific software system which has been deployed and is operational. A software system may have many releases over its life cycle, and would include all objects belonging to a project, but is normally a subset of a project.

Version - A version is a variation of a release. Different versions of the same release might be required to support multiple (but slightly different) physical installations of the system.

Increment - An increment is a management unit of a project associated with incremental development. There may be several increments leading to a single release, or an increment could equal a release after the system is operational.

It would simplify processing associated with macro relations if they could be represented in some hierarchic structure, but such a hierarchical ordering is not practical for macro relations. As an example, it would be convenient to associate a release of a software system with a project. However, in practice, many releases might be controlled by a single project and, for very large systems, many projects might be associated with a single release. Similarly, many releases of possibly different systems might use the same version of a baselined subsystem controlled by another project, and many versions of a system might be associated with one release.

Macro relations can be thought of as different views (possibly overlapping) of large pieces of the SEE data base. The many-to-many nature of these views again suggests that a relational approach might be better than a hierarchical approach when designing the SEE data base.

1.2.2 Micro Relations

There are many limited relations among data objects that should be supported by the SEE. These relations are called micro relations because they are significant as small sets of data objects. These micro relations are briefly described below.

Traceability - Traceability relates individual

requirements to the part of the specification, design, and source code which implement them, and to the data which demonstrates their correctness. This capability is useful in assessing the impact of requirements changes, and in defining regression test sets for changes to the baseline.

One method of attaining traceability is through use of the relational nature of the semantics tools defined for requirements analysis. These tools can be used to establish references to the specification, design, source text, and to test baselined data objects.

The SEE also shall provide for traceability directly through the use of micro relations. The traceability relation will allow a direct tie of requirements to baselined objects, or an indirect tie using an associative attribute as shown in Figure 1.2.2-1.

Derivation - Many data objects are related because they are derived from other data objects in the data base. In some cases, it is important to retain this relation explicitly to simplify the later use or modification of the derived data object. An example of such a situation is the inclusion of derived documents as part of the baselined products (see Figure 1.2.2-1). Another example is the binding of certain classes of measurements to the objects they measure.

Ancestry - Most data objects change during development and during the life cycle of a system. In many cases, such as source code of programs, it is important to maintain the sequence among predecessors. An ancestry relation shall be provided to allow for this.

Version - The version macro relation refers to the mutation of a system or system release. To make this view workable, micro relations should exist which identify different mutations of data objects. For example, certain programs might have several current implementations or versions.

Association - Certain data objects can be related in ways that should be identified explicitly. For

example, a performance model might have been developed for an alternative system design. It is important to be able to associate this model with the appropriate design.

The SEE support system should be capable of monitoring the data objects identified as having micro relationships. It should flag reprocessing when one data object within a micro relation is changed. An example would be a change in a design. The source text and related data objects would need to be examined and possibly adjusted.

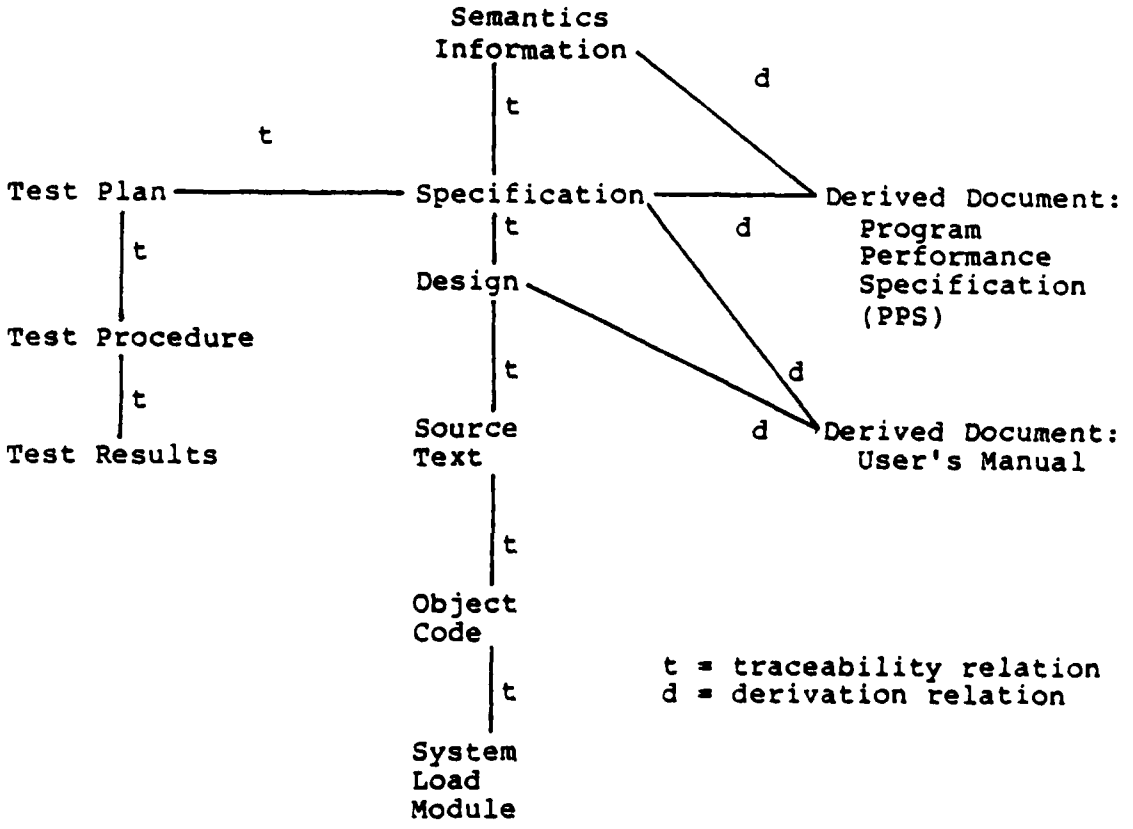


Figure 1.2.2-1: Examples of Micro Relations among Baselined Products

2. Support System

A support system is an integrated set of computer-based tools supporting all activities of the software engineering process. The term "integrated" implies a system where related tools have simple, compatible interfaces and a consistent structure across all tools. The support system contains a host computer and peripheral devices, extensive software, and special interface equipment.

The support system requirements presented do not constrain the choice of hardware or the hardware architecture. These decisions are reserved for the SEE designers. However, characteristics of I/O devices, target system interfaces, and performance constraints are given which restrict the degree of latitude in hardware selection. Some design constraints are defined for the software to ensure portability, architectural flexibility, and ease of change.

The support system (hardware and software) must be capable of handling all requirements of the standard military computers used by the Navy as well as future extensions to these computers. The Navy standard computers include the AN/UYK-43, the AN/UYK-44, the AN/UYS-1, the AN/UYS-2, the AN/AYK-14, the AN/UYK-7, the AN/UYK-20, and CP-642.

The support system must have a mechanism for creating a hierarchy of priorities between projects and between groups of users (or individuals) within projects. Creating and maintaining the priority structure is the responsibility of the management function controlling the critical resources of the SEE.

The requirements for the support system are presented in the following sections. External interfaces, functional capabilities, performance constraints, and design constraints are described. In each section, an introductory discussion is followed by itemized requirements. This format was chosen to improve traceability and correctness analysis for future implementations. Most of the itemized requirements are mandatory and are marked (R) for required. Optional characteristics of the support system are tagged by (O).

The sections under Support System vary widely in the

number of details presented. Because of the importance of the command language as the primary user interface, it is given detailed treatment. The text editor and the report generator are also given special emphasis because of their general applicability across activities. Special tools are not addressed in detail due to the treatment given them in Part I of this document.

2.1 External Interfaces

The external interfaces to the support system include the command language, I/O devices, and interfaces to target systems. The command language is important because it provides the only means to access SEE resources, and because of its importance an entire section is devoted to it. The characteristics of the I/O devices are defined only as necessary to ensure minimum capability. The universe of possible interfaces for target systems is given with general information provided for special devices supporting simulation/stimulation.

2.1.1 Command Language

The main objective of the SEE command language (CL) is to support software professionals by augmenting the creative, intellectual activities and automating the clerical activities. The CL provides access to SEE resources by SEE users - managers, designers, programmers, and testers. The CL is the only way to access the SEE data base and support system functions. It must be consistent across all SEEs.

Because of the wide range of user skills, interests, and use patterns, the CL must be human-engineered for both the casual and the sophisticated user. A simple, easy-to-learn subset must be available at one end of the spectrum. At the other end, the CL must have the characteristics of a high-level programming language for the user who must perform complex work assignments on the SEE.

Such a diverse set of users, accessing the SEE resources and representing many different projects, will require the CL to have a comprehensive and flexible set of access con-

trols. These controls must maintain the integrity of the SEE resources as well as protect the integrity of a project's or user's data. These controls will also provide management with a means for controlling access by personnel to baselined products and SEE resources.

2.1.1.1 Characteristics

The following sections list the required characteristics of the SEE command language.

2.1.1.1.1 Simplicity

The command language must have the attribute of simplicity.

Users (R) - The CL must support a range of users from the inexperienced, casual user to those who use the SEE for complex work assignments. For the inexperienced user, the CL must be capable of use in a simple, non-procedural, menu-driven form.

Prompting (R) - When used interactively, the CL must have a prompting mechanism to aid the inexperienced user; prompting must be explicitly requested to trigger it.

Defaults (R) - The CL must simplify its use by including defaults. Parameter defaults, as well as the ability to customize processing sequences should be available.

2.1.1.2 Power

The command language must also have power:

[illegible] - The CL must provide the capabilities of a high-level programming language for the sophisticated user. These must include conditional and loop control structures, argument han-

dling, variables, string manipulation, named procedures, and simple but powerful I/O instructions.

Additional Capability (O) - Additional programming capabilities should include control of stored procedures via parameters and simple but powerful arithmetic expressions.

Pipelining Capability (R) - The CL must support the concepts of "pipelining" and "redirection." Pipelining defines the standard output of one program as the standard input to another with both to be run in sequence. Redirection defines a non-standard destination for output upon error termination or normal completion of a program.

User Extension Exits (R) - The CL must be extendible through user-defined commands like functions in a high-level programming language.

2.1.1.1.3 Integrity

The command language must foster reliability:

Reliability (R) - The command language must be designed to foster creation of reliable programs.

2.1.1.1.4 Syntax

The syntax of the command language must have certain attributes:

Consistent (R) - The CL syntax must be consistent from its high-level, non-procedural use, down to its detailed use as a programming language. It must also be consistent across all SEEs.

ASCII Character Set (R) - The CL must use the ASCII character set.

Included Comments (R) - The CL must allow comments to be included in CL procedures.

English Structure (R) - The CL must be similar to English with a provision for abbreviated forms for the experienced user.

2.1.1.1.5 Entry Modes

The command language must support several modes of data entry:

Interactive (R) - The CL must be available in an interactive mode with prompting support.

Batch Control (R) - The CL must control the execution of batch jobs in the SEE.

Automatic Invocation (R) - A CL procedure must be capable of being invoked from storage as though it had been triggered from an interactive or batch mode.

2.1.1.1.6 User Aid

The command language must provide prompting information:

HELP (R) - "Help" must be available to the interactive user at any point during command entry. Help information must include the purpose, use, and format of commands and command arguments. The Help function must be designed to serve as an immediate source of information for a specific CL command or function.

Tutorial (O) - Tutorial support should also be provided to educate a user interactively in the basic capabilities of the SEE and the CL.

Abbreviated Commands (R) - Shortened CL commands and names must be provided to simplify data entry and reduce key strokes.

2.1.1.1.7 Exception Handling

The command language must be able to handle a number of exception conditions:

Messages (R) - Informative messages in English must be provided when a CL-related exception condition is detected. The messages would have to include errors related to syntax as well as content.

Specific Identification (R) - The specific command or argument causing the error must be identified.

Correction (R) - Correction of the error must be minimized to the data in error.

Abort or Redirection (R) - The ability to specify a redirection on a given error condition must be provided. Also, an abort on the error condition must be permitted; the abort must be graceful and the user notified.

2.1.1.2 Capabilities

The following sections list the SEE capabilities available through the command language.

2.1.1.2.1 Session/Job Control

The command language must control both sessions and jobs:

Sessions (R) - An interactive session between a user and the SEE must be started and ended through the CL.

Batch (R) - Batch jobs must be started and controlled during execution through the CL.

English Structure (R) - The CL must be similar to English with a provision for abbreviated forms for the experienced user.

2.1.1.1.5 Entry Modes

The command language must support several modes of data entry:

Interactive (R) - The CL must be available in an interactive mode with prompting support.

Batch Control (R) - The CL must control the execution of batch jobs in the SEE.

Automatic Invocation (R) - A CL procedure must be capable of being invoked from storage as though it had been triggered from an interactive or batch mode.

2.1.1.1.6 User Aid

The command language must provide prompting information:

HELP (R) - "Help" must be available to the interactive user at any point during command entry. Help information must include the purpose, use, and format of commands and command arguments. The Help function must be designed to serve as an immediate source of information for a specific CL command or function.

Tutorial (O) - Tutorial support should also be provided to educate a user interactively in the basic capabilities of the SEE and the CL.

Abbreviated Commands (R) - Shortened CL commands and names must be provided to simplify data entry and reduce key strokes.

2.1.1.1.7 Exception Handling

The command language must be able to handle a number of exception conditions:

Messages (R) - Informative messages in English must be provided when a CL-related exception condition is detected. The messages would have to include errors related to syntax as well as content.

Specific Identification (R) - The specific command or argument causing the error must be identified.

Correction (R) - Correction of the error must be minimized to the data in error.

Abort or Redirection (R) - The ability to specify a redirection on a given error condition must be provided. Also, an abort on the error condition must be permitted; the abort must be graceful and the user notified.

2.1.1.2 Capabilities

The following sections list the SEE capabilities available through the command language.

2.1.1.2.1 Session/Job Control

The command language must control both sessions and jobs:

Sessions (R) - An interactive session between a user and the SEE must be started and ended through the CL.

Batch (R) - Batch jobs must be started and controlled during execution through the CL.

2.1.1.2.2 Access Control

The command language must control user access to SEE resources:

Access Control to SEE (R) - An access control mechanism must be provided to protect key SEE resources. These resources must include executing SEE programs, SEE programs resident on the data base, and any other data in the SEE data base which is vital to the correct operation of the SEE.

Access Control by Project (R) - The access control mechanism must support certain management functions. It must be possible to restrict the access of project personnel to various classes of project data. The baselined products, in particular, must be completely under the control of configuration management.

Data Protection (R) - Individuals must be able to define data in their work space as private. This must not preclude management inspection of work-in-process.

Control of Security Classifications (O) - The access mechanism should provide for security classifications for designated portions of the SEE data base.

2.1.1.2.3 Data Base Manipulation

The command language must control the manipulation of data objects:

Data Base Creation (R) - It must be possible to create new data base objects through the CL facilities.

Data Base Utilities (R) - It must be possible to retrieve, copy, rename, store and delete data base objects through the CL.

Authorized Data Base Commands (R) - Commands for data base manipulation which the user is authorized to perform must be available to the user.

2.1.1.2.4 Program Invocation

The command language must control the use of programs:

Program Invocation (R) - The CL must be capable of invoking SEE programs or any other program in the SEE data base.

Interactive/Background Interface (R) - The CL must be capable of creating background jobs from an interactive session and report to the user the status or results of the job and the disposition of the data.

2.1.2 Characteristics of SEE Hardware

SEE's are very sophisticated systems. The importance of its hardware should not be underestimated as it has a significant impact on the quality and performance of the SEE to the user. The SEE must support many users with widely differing applications and requirements. It needs good, reliable hardware to fulfill its mission.

The SEE's processing power will not be discussed. The hardware it needs to support its users directly will be discussed but in terms of general characteristics. Detailed specifications for hardware will not be presented. The SEE hardware needed to support the user directly is listed:

Interactive Text Display (R) - This video device must support the display and entry of data interactively. It must display at least one page of alphanumeric data. It must be easily read and designed for minimum fatigue with continual use. It must have local memory equivalent to two full pages. It must allow remote correction to the SEE via a synchronous protocol. It must provide for upper and lower case, have a clearly visible cursor, support the 128 character ASCII set, contain special function keys, and allow selection of character or fields on the display.

Bulk Printer (R) - This device must be designed for high speed printing of entire alphanumeric text

lines. It must support the ASCII character set.

Interactive Graphics (O) - This device should be similar to the interactive text display but be able to display and manipulate vector graphics. The display will have a controllable cursor capable of addressing a single display picture element (pixel).

Bulk Text Entry (O) - This device should scan printed alphanumeric text and electronically encode them automatically. It should be capable of reading multiple fonts.

Bulk Graphics Entry (O) - This device should scan line graphics and electronically encode them automatically.

Quality Printer (R) - This device must create printed output of publication quality (i.e., camera-ready copy). It must support multiple fonts and simple line graphics.

Graphics Printer (O) - This device should create complex, high quality graphics of publication quality.

Individual Workstations (O) - Individual workstations, based on microprocessor technology, might be provided for users. These devices should be able to hold portions of the SEE data base temporarily, and to operate in either stand-alone or attached modes. They should provide the equivalent I/O capability of an interactive text device with printed output and, possibly, interactive graphics.

The SEE computers must support, either directly or via I/O controllers and communication controllers, the following physical interfaces: Electronic Industries Association (EIA) RS-232C, RS-422/423, X.21, IEEE 488, and optionally X.25. All I/O devices shall adhere to at least one of the required three physical interface standards. For parallel transfer, the American Standard Code for Information Interchange (ASCII) 7-bit-plus-parity code must be used.

User Class	Purpose	I/O Devices
Operators	Operation and control of SEE physical resources	Interactive Text Display (R) Bulk Printer (R)
Project Personnel	Development and continuing adaptation of ECS software	Interactive Text Display (R) Interactive Graphics (R) Bulk Text Entry (O) Bulk Graphics Entry (O) Bulk Printer (R) Quality Printer (R) Graphics Printer (R) Individual Work-stations (O)
Computer Science Analysts	Analysis of measurement and archive data to improve software engineering process.	Same as for project personnel

Figure 2.1.2-1: SEE User Classes

2.1.3 Target System Interfaces

The SEE must support interfaces to ECS target system equipment mainly for supporting software tests and ECS system tests. Various interfaces must be provided to the target systems on the assumption that the SEE provides for down-line loading of the ECS software, environment simulation, data extraction and reduction, and performance monitoring in support of tests.

The target system interfaces can be divided into two classes: computer-to-computer and sim/stim. The first class addresses computer-to-computer links between SEE computers and target computers. The second class addresses interfaces for stimulation of target system equipment.

The computer-to-computer interface will link a SEE computer to a target system computer such that each appears to be an I/O device to the other. The standard Navy military computers supported include the AN/UYK-43, AN/UYK-44, AN/UYS-1, AN/UYS-2, AN/AYK-14, AN/UYK-7, AN/UYK-20, and CP-642.

The stimulation interfaces for target system equipment will be supported through one or more interface controllers with the following characteristics.

Modular Interface - The interface equipment will be modularly constructed and programmable; this will simplify changes to handle the special interface requirements of new target system equipment.

Signal Conversion - The interface will provide for analog/digital, digital/synchro, and other conversions to support special target system equipment.

Format Conversion - The device will be capable of format conversions under program control.

2.2 Functional Capability

This section describes the functions of the support system. It is only concerned with the externally accessible

[illegible]

[illegible]

[illegible] system functions and software engineering functions. System functions control the resources of the support system, coordination among multiple interactive users, and the integrity of the data base. The software engineering functions include data base manipulations and processes directly supporting the software engineering activities.

2.2.1 System Functions

The support system must provide interactive and batch processing in an environment with many users. It will control all use of system resources, support local or remote users on a high-priority basis, and support batch processing on a low-priority basis.

2.2.1.1 Multi-User Interactive Support

The system must support multiple users in an interactive environment:

Multiple Users (R) - The system must support multiple users accessing SEE resources in an interactive mode; users may be either local or remote.

Resource Sharing (R) - The system must allow sharing of SEE resources; contention for limited resources shall be efficiently coordinated.

Multi-Level Security (R) - In the implementation of any of the system services for a SEE, no design decisions should be made that would preclude the implementation or retrofit of multi-level security operations.

2.2.1.2 Background Processing

The system must provide a foreground/background environment:

Foreground/Background (R) - The system must have both a foreground and a background mode thus taking advantage of the inherent nature of interactive processing. The system will allocate most of its priority resources to service the foreground, thus providing good response for short, interactive processes. Background processing must be provided to make full use of system resources. Stacked batch jobs will be started automatically by background as resources become available. (Batch jobs are sequential in nature and require no manual intervention.) Batch jobs can also be run in foreground.

Batch Jobs (R) - It must be possible to enter jobs on a queue from an interactive session without terminating the session. Notification of the job status, and access to the results during the interactive session must be possible.

2.2.1.3 Data Base Management

The system will provide full data base support:

DBMS (R) - The data base management function (DBMS) must allow creation, retrieval, and modification of data identified in Section II.1.1, and must include data relations.

Logical Views (R) - The data base management function must support logical views of data base objects as identified in Section II.1.2.

High-Level Functions (R) - The data base management function must allow manipulation of the data base through high-level function calls which require no knowledge of the physical organization or location of the data.

Micro Relations (R) - The data base management system must support micro relations and the maintenance of these relations under changes to data objects.

2.2.1.4 Pipelining and Redirection

The system must support pipelining and redirection:

Pipelining (R) - Pipelining permits several programs to be run in succession without operator intervention. The output of one program can be designated as the input to one or more succeeding programs via temporary files or buffers.

Redirection (R) - Redirection of output due to abnormal error conditions or termination of a job must be supported.

2.2.1.5 Operational Control

The system must provide the system operator with a full set of controls:

Control of Support System (R) - Control must be via the on-line operator's console. The operator must have adequate facilities and flexibility to control how, when, and to what extent, system operations are performed. Functions such as system startup, monitoring of operations, and system termination must be available. All system functions must be capable of operation in full, partial, or degraded modes.

System Configuration (R) - The system operator must be able to configure the system during startup or during operation. Reconfiguration shall be non-disruptive to system users. (Configuration changes can be made for a number of reasons such as a component failure, the need for preventive maintenance, or an intermittent error condition.)

Backup and Restore (R) - The operator must have a

simple, flexible mechanism for saving and restoring data. The mechanism must be able to perform full or partial backups or restores on a selective basis. It must be able to recreate data from a backup image and modify it with the transactions occurring since the last backup.

Short-term and long-term backup storage must be available, the first for active data and the second for inactive data. The operator must be able to restore active data quickly and inactive data simply and directly. Restoring inactive data will require search arguments (such as project identifier) to locate the data before it is restored.

System Degradation (R) - The support system must be able to degrade gracefully under increasing levels of system errors. System users must be provided either automatic or semi-automatic save procedures so that work-in-process can be saved before a forced or unforced sign-off. The system must log the status for later analysis at the point of degradation or crash.

System Recovery (R) - The operator must be able to restore the system automatically, either fully or partially, after system degradation or a system crash. The operator also must be able to determine the system configuration, and the status of the background job stream, at the time of the degradation or system crash.

System Statistics (O) - The support system should keep adequate statistics on what resources have been used and by whom for analysis, cost accounting, and billing.

2.2.1.6 Remote Access

The system must support remote users:

Remote Users (R) - Users who are remote geographically must be able to access SEE resources over telecommunications paths for either interactive or

batch operations.

Remote User Interface (R) - The interface must be the same for remote users and local users.

2.2.1.7 Access and Resource Control

The system must provide certain control functions:

Access Control (R) - A system mechanism must be provided to control access to the SEE via a system-wide list of authorized users. Access control must also include password protection.

Resource Control (R) - A system mechanism must be provided to control access of authorized users to SEE resources. Resource control must also include password protection.

Data Protection (R) - A system mechanism must be provided to permit data to be protected as read-only data, read/write data, or update data (read implied). It must be possible to protect data (files, records or fields) via a password mechanism. Such protection shall not limit management control.

Security Protection (R) - A system mechanism must be provided for protecting all or part of a data base containing classified information. Such a mechanism must also be capable of protecting project information separately. (Encryption is one technique for protecting information either partially or fully. Different cryptographic keys can be used to protect information from different projects or different users or both).

2.2.1.8 Utilities

The system must provide a variety of general-use functions:

SEE Data Base Transport (R) - The SEE must provide

the tools needed to transform baselined products into the Navy standard form; such data can then be input to another Navy standard tool and vice-versa.

Support System Utilities (R) - The SEE support system must provide the standard utilities needed to manipulate the data base and files. Such standard utilities must include (for a variety of I/O devices), sort/ merge, move/copy, report generation, retrieval capability, data base query, data dictionary, a data base language, and others.

2.2.2 Software Engineering Functions

The software engineering functions of the support system span all of the activities of the software life cycle defined in Part I. The functions defined under "general services" are applicable to all activities. Those under "activity-specific tools" are applicable to a subset of the activities. "Application paradigms" addresses the means of capturing software components and designs for later reuse.

2.2.2.1 General Services

The general services [illegible] aid the software engineering activities but are not specific to any one activity or subset of activities. The general utility functions [illegible] text editor, document generation, and [illegible]

[illegible] Text Editor

[illegible] shall support the creation and modification of text in an interactive text terminal using either a full-screen or a line-oriented editor. The physical devices supported shall include those with the characteristics defined for an interactive text terminal in Section II.2.1.2.

Full-Screen Editing (R) - The text editor must support full-screen editing on a full page of text as displayed on the interactive text display.

Buffering Capability (R) - Edit operations must be performed on text in an edit buffer and will have no effect on the data base copy until a store operation is explicitly given. Text in the edit buffer will be viewed on the display through vertical and horizontal scrolling commands. It must be possible to merge separate text objects in the edit buffer.

Split-Screen Mode (O) - Multiple edit buffers should be assigned to a single terminal user. It should be possible to view these through split screen operations; each edit buffer would be acted upon as though it were supporting a single terminal.

Text Composition Mode (R) - The text editor must support a text composition mode with the following minimum operations:

- Continuous typing with word wrap,
- Automatic reflow of text when column boundaries are changed, and
- Insert and delete operations which exceed single line boundaries.

Character String Substitution (R) - The system must provide for the substitution of one character string for another, wherever it appears within a designated block of text, or within the edit buffer in its entirety.

Tab Control (R) - It must be possible to define tabs on the screen to simplify cursor movement.

Cursor Positioning (R) - Control of the cursor positioning from the terminal must permit up, down, left, and right movement, and commands for "home."

Edit Command Sequences (O) - The system should permit named sequences of edit commands to be stored. These sequences should be invoked to operate on

data placed in the edit buffer.

Consistent Syntax (R) - The syntax and conventions of the edit commands must be consistent with the command language syntax.

Structured Progamming (O) - The editor should support structured programming techniques by automatically indenting PDL and code, according to the nested use of control structures, and automatically underscoring PDL keywords.

Highlighting (O) - The editor should use highlighting (or reverse highlighting) to emphasize text elements which are subjects of certain operations such as an error condition.

Line-Oriented Editing (R) - The text editor must provide certain line-oriented operations: line numbers for use in edit operations; a "find" command to locate a line number or character string; commands to insert, delete, move, or copy lines or blocks of lines; commands to shift lines or blocks of lines left or right; and commands to insert, delete, or replace characters within a line.

2.2.2.1.2 Document Generation

The system must provide a general purpose report generator:

Formatting (R) - The document generation function must format text for viewing on an interactive text device or for printing or plotting graphic material.

Photocomposition (O) - Text should be formatted for photocomposition equipment (or equivalent).

Input for Document Generation (R) - The input function must permit: text, symbolic codes representing strings to be substituted in the text before printing, embedded commands which control text format, and macro commands. (Macro commands are

user-generated super commands consisting of groups of embedded commands.)

Page Layout (R) - Certain page layout functions must be provided:

- Line formatting including concatenation, centering and right and left justification;
- Definition of blank lines for spacing;
- Definition of paragraph style and spacing;
- Provision for multiple fonts designated by running headings or text blocks;
- Column definition;
- Definition of top, bottom, left, and right margins;
- Provision for running headings and footings, and automatic page numbering; and
- Definition of tab settings for tabular information.

Text Composition - Certain text composition functions must be provided:

- Automatic flowing of text into columns for proper width, length, and layout specifications;
- Provision for inseparable text;
- Hyphenation;
- Provisions for text blocks which must start at the top of a page or the top of a column.

Headings - Up to seven levels of heading must be allowed, each with unique formatting specifications.

Table of Contents - Automatic generation of the table of contents with page numbering must be pro-

vided. Optionally, manual placement shall be allowed.

Footnotes - Floating footnotes must be placed at the bottom of the appropriate pages or optionally placed at the back of the document.

Revision Indicators - Provisions must be made for indicating revisions on each page.

Dictionary - Provisions must be made to provide a function for checking spelling of standard English as well as project-specific terms, acronyms and mnemonics.

Index and Cross-Reference - Provisions must be made for automatic generation of terms flagged for an index or cross-reference.

Standard Syntax - The syntax and conventions used for the interaction with this tool must be consistent with those defined for the CL.

2.2.2.1.3 Tool Building Aids

The system must encourage the creation and use of tools constructed from tool atoms (tool fragments). Tools will always be evolving as old tools are extended, new tools are created, and single-use tools are quickly constructed and discarded. The tool building mechanism should be simple, flexible and efficient to use.

Tool Atoms (R) - The SEE must provide a set of tool building blocks (tool atoms). These atoms alone perform no useful tasks but in combination with other atoms can create useful software engineering tools. The minimum tool set must support conversion and transit use functions.

[illegible] involves the restructuring of application software. Translation applies to existing application software [illegible] new support system and host.

Tool atoms will be designed for, and mostly used by, the software engineering specialists. This approach limits the general application of tool atoms but also simplifies the design and interface structure since tool atoms need not be designed for universal use. This does not preclude the use of tool atoms by application programmers (and others) but they must be willing to learn how to design and construct tools from atoms.

Combining Atoms (R) Tool atoms must be designed for combination in pipelining fashion, or as called procedures within a command language program.

Index (R) - An index of abstracts describing available tool atoms must be provided. This index would be used during tool design and implementation.

2.2.2.2 Activity-Specific Tools

Requirements for activity-specific tools are derived from the discussions of activities in Part I. These discussions outlined the methodologies which lead to the need for automated tools. This progression from activity to methodology to tool is summarized in Figure 2.2.2.2-1. Descriptions of each tool can be found in Part I under the appropriate activity description.

Only a small set of the tools is mandatory and marked (R) for required. These are the tools required for the formal recording of baselined products plus some frequently used analysis tools. All other tools are marked as optional (O). This approach is in line with the strategy of rigorously defining the baseline product methodologies, but allowing the professional freedom of choice in the supporting methodologies and tools which augment analysis and creativity.

Each of the tools in Figure 2.2.2.2-1 to 2.2.2.2.6 should possess certain basic characteristics to assure integration within the SEE support system. These characteristics are:

Standard Capabilities (R) - Each activity-specific

tool must use the standard capabilities of the SEE operating system and data base manager.

Consistent Syntax: Mandatory Tools (R) - Each activity-specific tool which has been identified as mandatory, and which includes a command language, must be implemented such that its command language syntax and conventions are consistent with the SEE command language across all SEEs.

Consistent Syntax: Optional Tools (G) - Whenever possible, optional activity-specific tools which include a command language should be implemented so that their command language syntax and conventions are consistent with the SEE command language.

High-Level Language (R) - Each activity-specific tool must be implemented in a high-level language to ensure portability.

Pipelining (R) - Certain groups of activity-specific tools must be implemented in such a way that pipelining is possible among them. The sets of such tools that must have pipelining capability within the set are shown in Figure 2.2.2.2-7. The arrows ==> indicate the sequence taken by pipelining, while the braces [ ] define the set of tools in the pipelining sequence.

Instrumentation (R) - Activity specific-tools must be instrumented to generate the measurements data described in II.1.1.3.

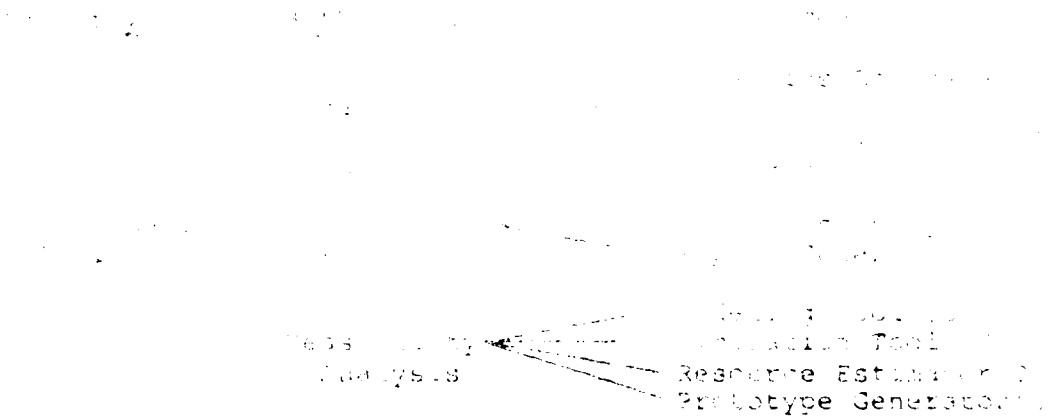


Figure 2.2.2.2-1: Flow of Requirements Analysis:
Activity to Methodology to Tool

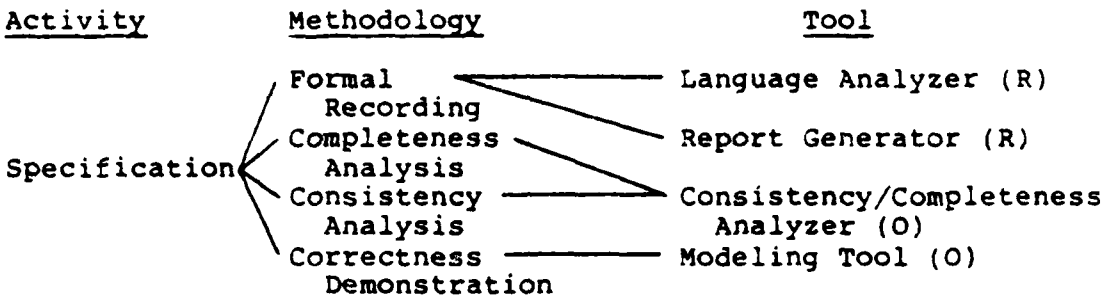


Figure 2.2.2.2-2: Flow of Specification: Activity to Methodology to Tool

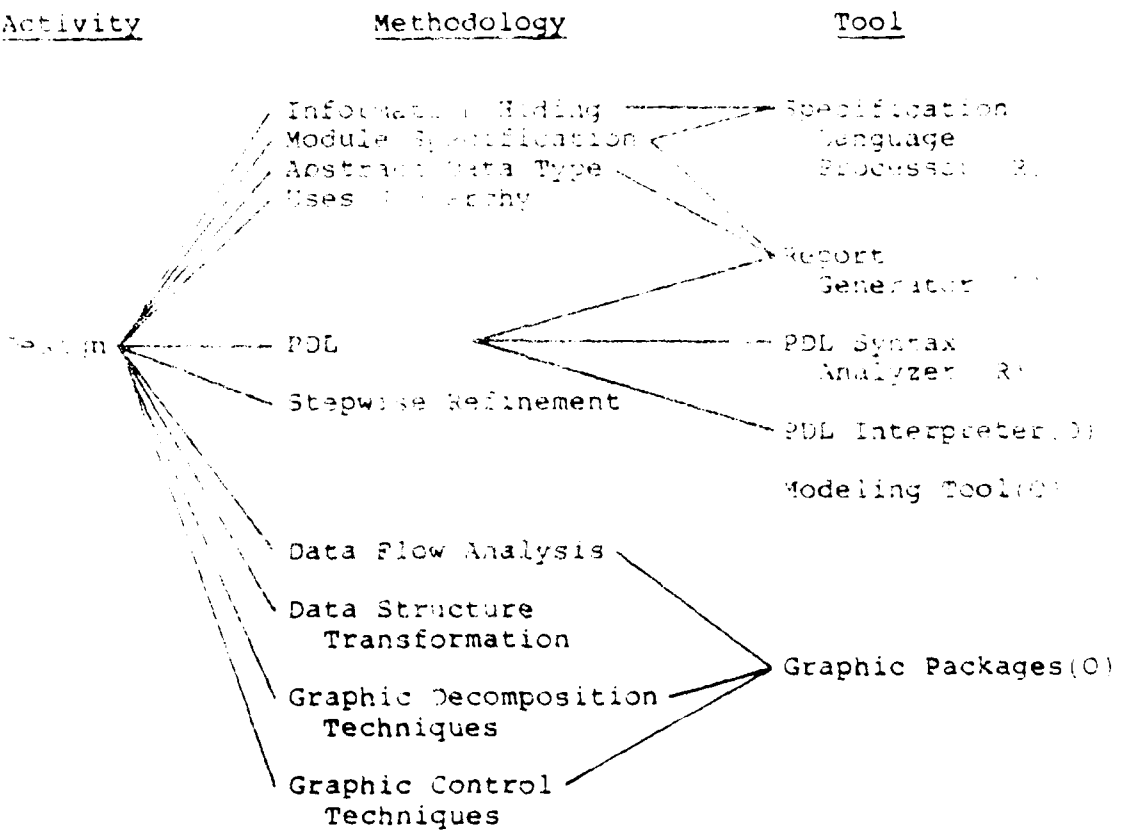


Figure 2.2.2.2-3: Flow of Design: Activity to Methodology to Tool

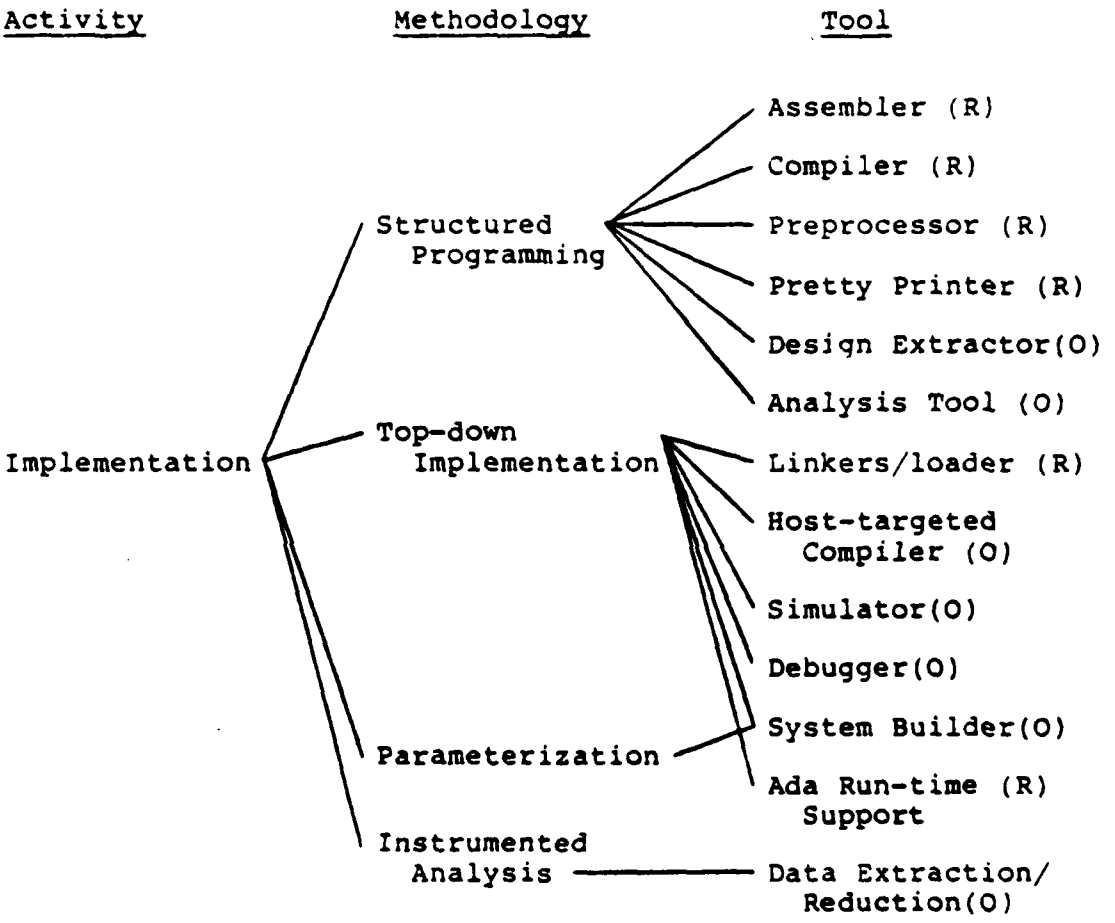


Figure 2.2.2.2-4: Flow of Implementation: Activity to Methodology to Tool

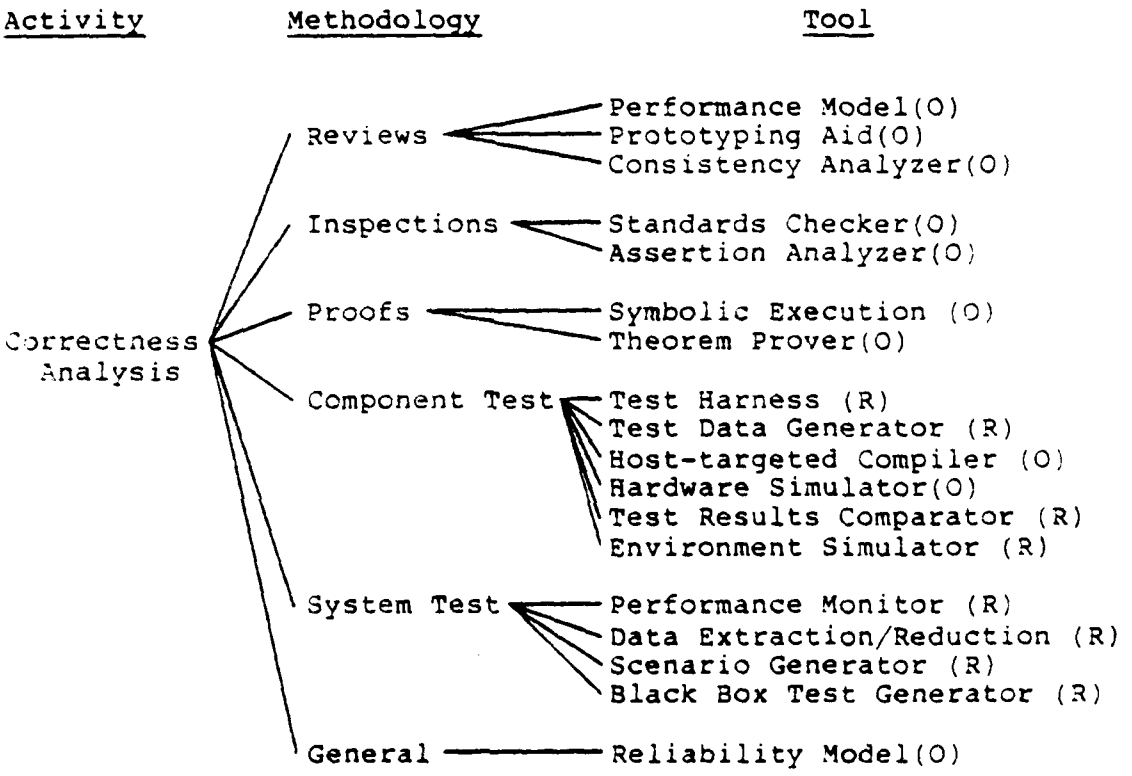


Figure 2.2.2.2-5: Flow of Correctness Analysis:
Activity to Methodology to Tool

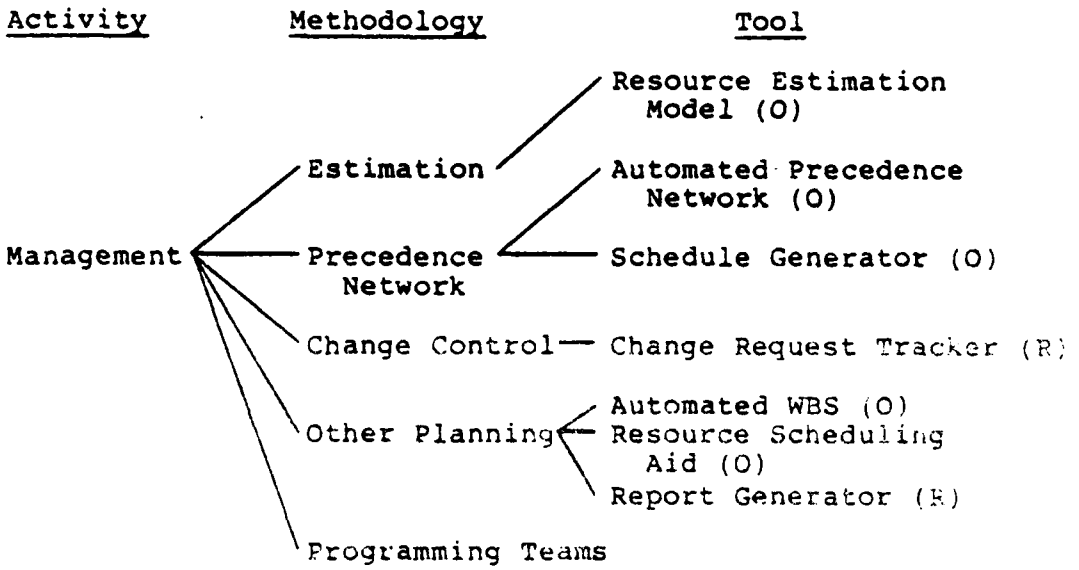


Figure 2.2.2.2-6: Flow of Management: Activity to Methodology to Tool

Requirements Analysis:

[Semantics language processor
==> semantics information browse
==> semantics analyzer
==> report generator]

Specification:

[Language analyzer
==> consistency/completeness analyzer
==> report generator]

Design:

[PDL syntax analyzer
==> PDL interpreter
==> report generator]

Implementation:

[Preprocessor
==> assembler or compiler
==> pretty printer
==> design extractor
==> analysis tools
==> compiler or assembler
==> linker/loader
==> hardware simulator
==> debugger
==> data extraction/reduction]

Correctness Analysis:

[Test data generator
==> test harness
==> hardware simulator]

Figure 2.2.2.2-7: Pipelining for Activity-Specific Tools

2.2.2.3 Application Paradigms

Many software design and implementation problems are well understood and continually recur in new systems. Unfortunately, the designs and implementations of previous systems are not captured and reused in succeeding systems. Thus, a significant opportunity for productivity improvement is lost.

Two features of the SEE are described to address this serious problem: reusable components and program skeletons. Reusable system components are commonly recurring components which can be designed and implemented once with appropriate parameters for tailoring to different applications; these components might include firmware. Program skeletons provide the structure into which application-specific modules can be incorporated.

2.2.2.3.1 Reusable Components

The system must support and encourage the creation and maintenance of reusable components:

Interface Standard (R) - A rigorous interface standard must be provided for all candidate reusable components; it would govern design, implementation, and documentation.

Library (R) - A library of reusable components must evolve for use by all development activities.

Index (R) - An index of reusable components including summary descriptions and specifications, must be provided to help the designer locate appropriate components.

2.2.2.3.2 Program Skeletons

The system must also support program skeletons:

Design Standard (R) - A rigorous design standard

must be provided for any major software system or subsystem to be implemented as a skeleton.

Skeleton Library (R) - A sparse library of subsystem or system skeletons must be provided for use in new development activities.

Supporting Tools (R) - A set of supporting tools must be be implemented to aid in the expansion of the skeletons.

2.3 Performance Guidelines

The performance of a SEE plays an important role in operations--how well it responds to users, how many projects it can handle, and how reliably it performs. The basic capacity requirements, when overlaid on the functional and interface requirements, will determine the type and size of the system. The selection of hardware, software and data base are all influenced by these requirements.

The size of a SEE cannot be stated definitively as it depends on the installation. Sizing must consider factors such as: the number of projects supported, the size of the projects, the number and size of simulations, and the user population. These factors will influence the size characteristics: number and power of processors, memory, on-line data storage, number of I/O channels, number of user terminals, and off-line data storage. These sizing factors also influence the support software: operating system, utilities, access methods, communications protocol, and unique software requirements.

The SEE must be planned for no more than 60% of the available capacity for normal operations. This permits more growth of the SEE during its early years. The SEE must be capable of a 200% growth in capacity through a natural, design-consistent expansion of the hardware.

An effective SEE must provide a stable environment for the users. This means not only high reliability (i.e., infrequent failures) but high availability (infrequent periods of extended outage). Assuming that reliability is high, availability can be kept high by having a well-designed

modular system and by having contingency plans for unexpected repairs or failures.

Since systems can fail in many ways, several categories of failures would be:

Redundant component failure: failure of a support system component for which alternate or backup units are available (e.g., an alphanumeric display).

Partial system failure: failure of a component for which backup is not available, resulting in restricted operation of the support system (e.g., failure of a channel).

System failure: any failure of a key system component that renders the entire support system inoperable and for which no backup is available.

Software can also fail:

Environment changes: software environment changes can make a user inoperative. For example, if the system software has been altered in some way and each user has been requested to make a corresponding change, a user can be disabled if he fails to make the change.

User-specific software errors: a user can be running a unique piece of software which can fail, be ineffective only that user does, and affecting no other users.

System errors: a system component can contain a fault which causes it to fail when the proper conditions exist; such a failure can be intermittent or consistent, and can cause all or part of a system to become inoperative.

The SSE must be capable of reliable performance with a minimum of unavailable time. The [illegible] the SSE must have [illegible]

2.4 Design Constraints

Several SEE design constraints are mandatory: it must be rehostable (capable of being moved easily from one host to another host computer), it must have ease of extension, and it must support Ada.

A SEE is rehostable if it can be moved to a different host computer and become quickly operable with the installation of the host-unique kernel. The ability to rehost a SEE can only be achieved through extensive use of a high-level language, and through keeping machine-dependent code within the non-transportable kernel. All code outside the kernel must be independent of the hardware and the operating system. Tools must be completely portable.

The attribute of being extendable can be attained through a flexible architecture and a standard interface for new tools. The flexible architecture allows tailoring to a variety of hardware architectures. New tools can be added to the support system if they comply with minimum interface standards.

2.4.1 Independence

The SEE design must support independence of software in the following areas:

Portable SEE (R) - The support system software must be portable except for the machine-dependent kernel. The abstract interface to kernel services must be identical across all implementations of the support system.

I/O Interfaces (R) - All I/O devices must be interfaced through abstract device interfaces to minimize the impact of changes.

Data Base Independence (R) - The support system software, with the exception of the data base management system, must be unaffected by the physical organization or the location of the data base.

Data Base Design (R) - The logical and physical designs of the data base must be strictly separated

[illegible]

2.4.2 Architectural Flexibility

The SEE must be flexible:

Adaptable Architecture (O) - The support system software should be adaptable to a distributed mini-computer architecture, or to a large mainframe architecture, or to a hybrid architecture.

Geographic Distribution (O) - The support system software functions should be adaptable to geographic distribution. This might be accomplished through system generation changes alone.

2.4.3 Ada Compatibility

The SEE must support Ada:

[illegible] (O) - The SEE must support a standard Ada compiler.

MAPSE Consistency (O) - The SEE must be consistent with the MAPSE as defined in the Top Level Requirements for the Navy MAPSE.

2.4.4 Generalized Tool Interface

[illegible]

standard SEE system services.

Information level (3) - The tool must use data [illegible] the SEE data base and control [illegible] SEE access mechanisms.

[illegible] - [illegible] be consistent [illegible] graphics [illegible] applications [illegible] describing [illegible] characteristics.

[illegible] level 4(?) - The syntax and semantics of tool [illegible] be consistent with the SEE command language. Help and error response mechanisms must be consistent with those provided by the SEE command language. (Required for baseline recording, otherwise optional.)

APPENDIX A

Some Background on

Software Development

significantly affected if software causes delays in the introduction of new systems or malfunctions in the weapon system. Either situation is likely, given the stresses on the software production resources and the availability and quality of the software engineering practices and tools.

1.1 A Comparison of Two Systems

This section compares two operational systems to show the disparity in software engineering practices in the Navy. The two systems are not ECS but are included under the MIL-STD-1679 umbrella.

Figure A.1.1-1 contrasts the software engineering practices and associated information for each system. The most striking attribute of the contrasted systems is that each is like a snapshot of the software engineering practices accepted as the norm during their development. System A reflects the middle 1960's, and System B the early 1970's. The information, based on a 1980 review, shows that software systems have an inertia which resists change. The cause may be a lack of clear requirements, or insufficient direction or motivation for the maintainers to re-engineer evolving software. Even with properly motivated maintainers, software engineering environments for existing systems are not designed to encourage reengineering.

Although these systems are but a small sample of Navy software systems, it is useful to state some generalizations based on them. These generalizations are corroborated by other Navy groups and systems.

Application of Standard - Current standards are not uniformly applied to Navy systems, and are not evident at all in older systems. Systems seem to persist in their original state. Re-engineering to improve the the software relative to standards is uncommon over the life cycle of software.

Support Tools Hosted on Target Computers - In many cases, the support tools are hosted on the target computer. This pervasive practice has been detrimental to the software. The target computers are constrained in power and resource, and they do not have a good set of modern software engineering tools.

1. State-of-the-Practice in the Navy

[illegible] the early 1960's, which occurs are using the technology of the late 1970's. This disparity reflects, to some degree, the sophistication of the group responsible for either the original development or current maintenance. Another factor is the pervasive practice of hosting development and maintenance on the target computer. The restricted power, resources, and support facilities of most existing target computers, in terms of software engineering, constrain what can be done and what is available.

Although the Navy has explicit standards for the development of ECS software, the standards are unevenly applied. Much of the ECS software inventory was developed before the adoption of MIL-STD-1679, therefore, the systems in the ECS inventory show varying degrees of adherence to current standards. For systems undergoing a major upgrade, using current standards to develop the modified portion of the system is often cost-justified. This process of upgrading the design and implementation of evolving systems will be called re-engineering. Re-engineering older Navy software to current standards is slow or non-existent in most cases.

The Navy has many software support systems and development tools available. These tools are good, but they address only a small segment of the software development process: code generation and test. Software production demands are rapidly increasing in the face of a modestly expanding programmer population. Thus, this support cannot improve productivity enough to bring projected demands into balance with projected resources.

The annual budget for Navy software is hundreds of millions of dollars, so the cost to the Navy for inefficient development and maintenance is huge. The avoidable cost should provoke concern about improving the ECS software engineering practices; however, it is not the most important [illegible] reflect readiness is even more important.

Software has become more significant with each new generation of weapon system. Fleet readiness can be

	SYSTEM A	SYSTEM B
Date Operational	1970	1977
Target Computer	CP901	PDP 11/70
Age of Target Computer	@16 yrs.	@7 yrs.
Host Computer	CP901	PDP 11/70
Software Architecture	Outmoded OS, change difficult, no hierarchic structure	Modern OS, reasonable modularity, structured
Standards	None evident	Partial adherence to 1679
Modern Methodologies	None evident	Structured programming Warnier-Orr design Formal T&E
Documentation	Minimum documentation, non-standard	Good documentation, non-standard
Support Tools	Card-based editing Tape libraries SYMON/SYCOL compiler Assembler, system integrator, link-11 simulator	On-line editing of disk-based libraries Good code generation support

Figure A.1.1-1 A Comparison of Two Operational Systems

Reinvention of Software - In all ECS systems today, many common functions can be identified. Nevertheless, these functions are designed and implemented anew, again and again, for each system. Development of reusable components could strongly affect productivity and reliability.

Lack of Management Rigor - Rigorous planning, configuration management, and quality assurance techniques are not consistently applied to most software development or maintenance projects.

Lack of Sufficient Correctness Analysis - Correctness analysis and software quality are pursued through testing alone. The necessary techniques to ensure high quality are not used on most projects.

1.2 Currently Available Support Systems

Support Systems: The Navy has several support systems for the development and maintenance of ECS software.

Machine Transferable AN/UYK Support Software - MTASS is a software support tool for Navy standard computer systems. It consists of a CMS-2 cross-compiler, a MACRO cross-assembler, a loader, a system generator, simulators, and a FORTRAN cross-compiler. MTASS provides library support for application programs and can be hosted on several different computers.

Facility for Automated Software Production - FASP is a support system, developed by the Naval Air Development Center (NADC). It is hosted on both CDC and PDP mainframes. FASP supports such languages as SPL/1, CMS-2, MACRO, and ULTRA; and it produces code for the AN/AYK-14, AN/UYK7, AN/UYK-20, AN/UYK-32, and AN/UYS-1 computers. It features an integrated data base containing programs as well as project management information. Tools available include a text editor, compilers, assemblers, linkers, loaders, target computer simulators, unit test tools, and management reporting aids.

SHARE/7 - SHARE/7 is a multiprocessor timesharing system hosted on the AN/UYK-7 computer. It provides

an interactive environment for the software development. It includes a text editor, on-line debugging aids, compilers, assemblers, loaders, target computer simulators, and a text formatter. The language processors generate code for the AN/USQ-20, AN/UYK-7, and AN/UYK-43 computers. SHARE/7 supports CMS-2, FORTRAN, SPL, BASIC and several assembly languages.

These three Navy software support systems are well-conceived, useful systems, however, they are limited since they address only code generation and unit test. The important functions of requirements analysis, specification, and design are inadequately supported. Correctness analysis also lacks support. There is a real need to extend support beyond the scope of these systems.

Simulation/Stimulation Systems: Another type of support system required for ECS software development and maintenance is an environment simulation system. These systems are sometimes called simulation/stimulation systems or simply sim/stim. The Navy has developed and operates many of these systems today for training and testing. A sim/stim system might consist of (1) one or more computers to host the environment simulation software, (2) special devices to interface the environment simulation computer to the ECS or other tactical hardware, and (3) a configuration of the ECS along with some subset of tactical hardware.

Although the Navy's sim/stim systems have many common functions, each sim/stim system has been developed with no attempt to create a set of reusable components. Since 15 to 20 (or more) such systems exist, each costing from $10 to $30 million to develop (plus more for maintenance), the Navy's cost for this redundancy is significant.

1.3 A Case Study

A functional analysis of one Navy software production group was performed as part of a software automation program. This functional analysis shows some of the problems and dynamics faced by Navy software production groups today. The following observations are derived from the final report which was prepared by Systems Consultants, Inc., October 15, 1980.

Figure A.1.3-1 vividly illustrates a common state in software production groups today[1]. Most of the automated resource is dedicated to code generation and test, with little or none of the resource supporting the activities of specification and design.

The functional analysis data shows a very disturbing trend of rapidly increasing workload in the face of constant or decreasing resources. Figure A.1.3-2 shows the projected work load between 1980 and 1990 against the available work force (based on current budgets). The imbalance is overwhelming. Even if the budgets could be increased, the facility probably could not hire the qualified programmers because of the projected programmer shortage.

A similar problem exists for the support systems and mockups used for testing. Figure A.1.3-3 shows the additional systems which would be needed to support the projected work load. These additional systems represent a significant capital investment. They will also require floor space not available at the facility.

The functional analysis attests strongly to the problems facing the Navy in ECS software. Accelerating growth in the workload will soon overwhelm the production resources and thereby threaten fleet readiness. Increasing the production resources to handle the work load is not feasible because of the cost and projected shortage of qualified programmers. The only answer is to increase the productivity of the available work force. This cannot be done w thout a significant improvement in the support systems, methodologies and tools used by these groups. The current support systems are too narrow in scope to meet future needs.

2. Modernization and R&D Activities

Many people in the Navy have recognized the need to improve the way software is developed and maintained. Several Navy activities, aimed at developing improved methodologies, tools and support systems, have alr ady begun. In the past, these activities were not coo dinated,

---

[1] The figures in this section are copies of charts prepared by W. Rosen of NOSC.

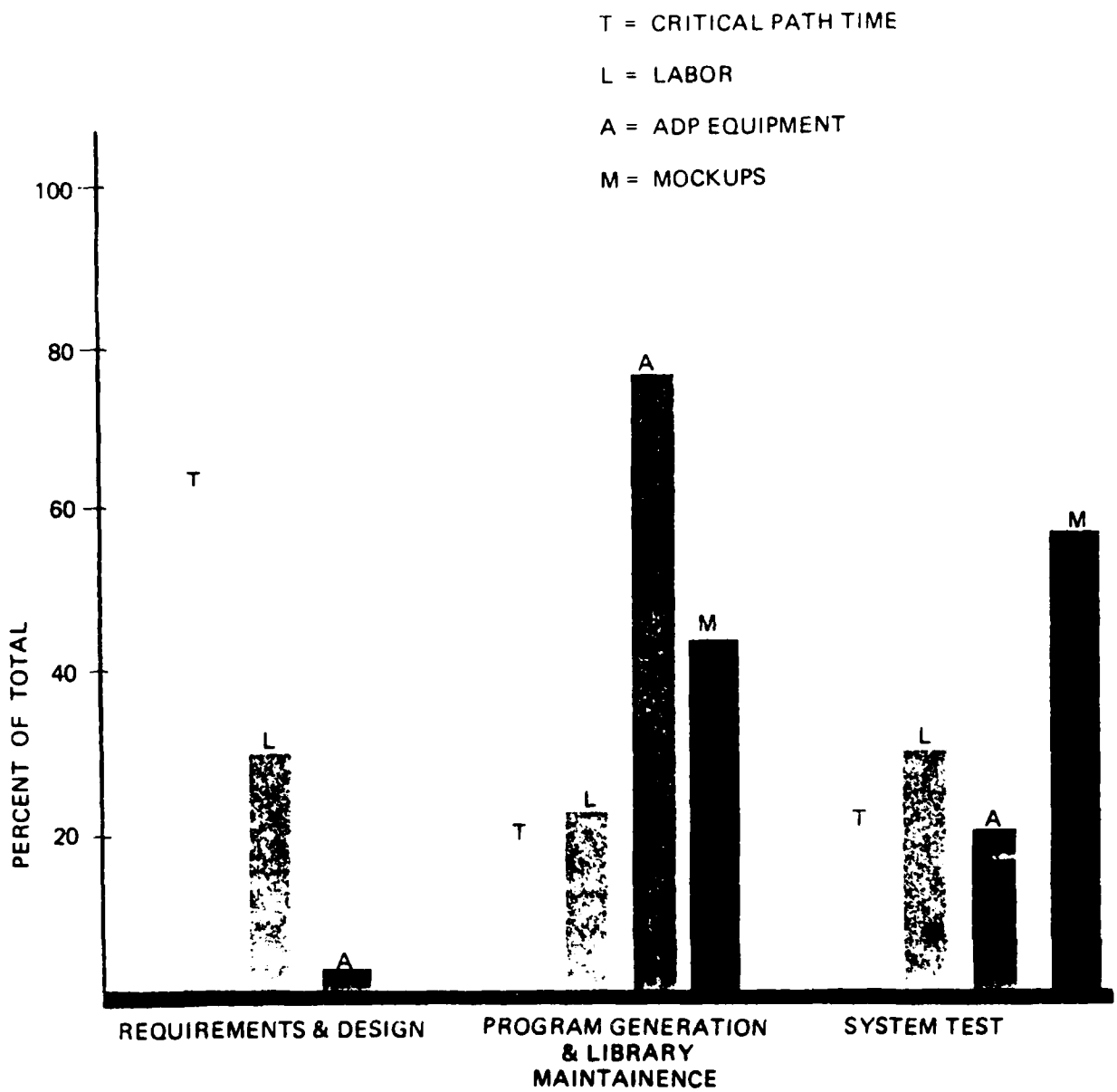


Figure A.1.3-1: Application of Automated Support

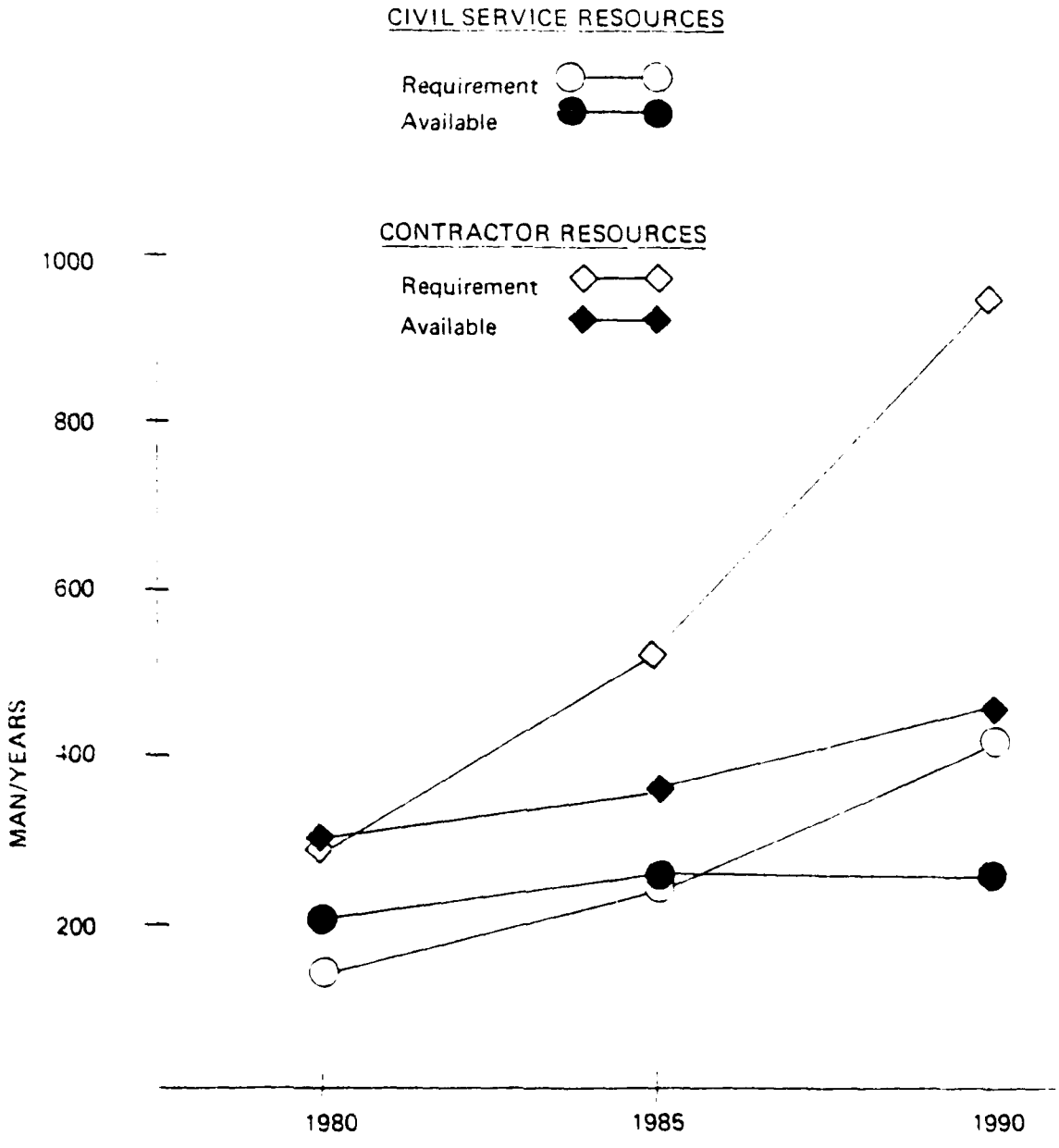


Figure A.1.3-2: Workload vs. Labor Resources

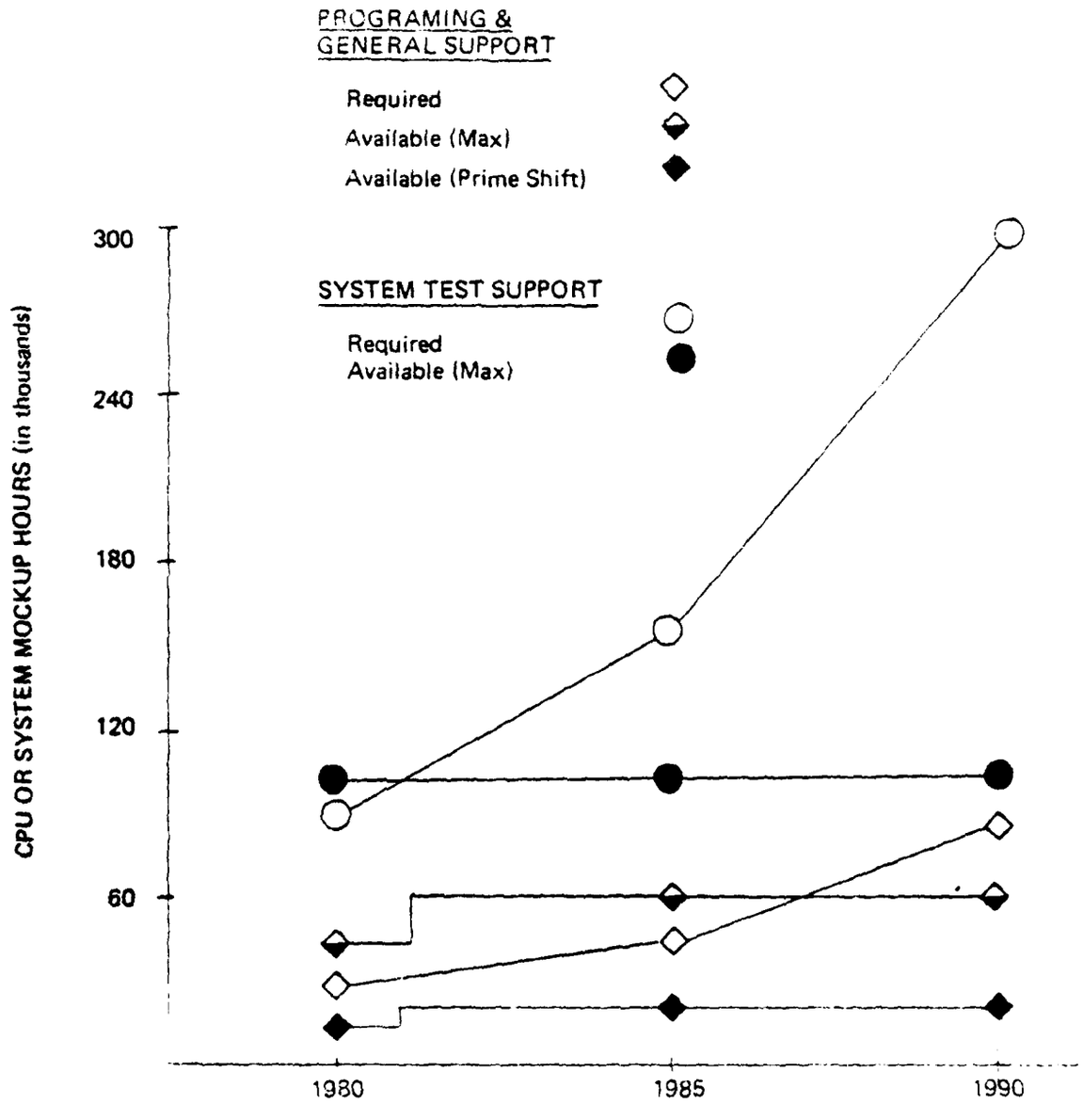


Figure A-1 [illegible] vs ADP Equipment Availability

and they often duplicated one another. Recently, the Navy has taken two significant steps to provide better focus for modernization of software engineering.

Coordination of Planning for Exploratory Development - This activity is part of Program 6 within the Navy's RDT&E accounts (Research, Development, Test and Evaluation) and is referred to as "6.2" resource. Within the NAVAIR, NAVELEX and NAVSEA system commands, people involved in 6.2 planning and control have begun a cooperative effort. This effort will produce a coordinated 6.2 plan for software engineering. It should provide a better focus for 6.2 resource and a more effective investment of the resource.

Software Engineering Environment Working Group (SEEWG) - MAT-08Y established the SEEWG to recommend ways to eliminate redundant development activities for support systems, and to focus efforts on developing a modern Navy SEE. The SEEWG includes members from all Navy system production groups and system centers involved in the development and maintenance of ECS software.

The results of these efforts will not be evident for some time because the number of modernization activities is large. The following subsections provide a sampling of these activities.

2.1 Modernization of Support Systems

In a preliminary survey of its members, the SEEWG identified more than 20 Navy modernization efforts in software development or sim/stim systems. The list is far from complete and does not include similar efforts by Navy contractors to upgrade their in-house capabilities. These modernization activities include developing project-specific support systems, as well as extending MTASS, FASP, and SHARE/7 described in Section 1.2. Summarized below are two important examples of modernization efforts.

TRIDENT Software Support System - TSSS is a support system recently installed for development and maintenance of TRIDENT ECS software. Hosted on two IBM 4341 computers, it uses many standard IBM software

packages for library control, text editing, document production, etc. An attached AN/UYK-7 compiles CMS-2 code and runs unit tests. TSSS supports code generation, unit test, and documentation.

NAVELEX Software Support Facility (SSF) - [illegible] has submitted a Program Objectives Memorandum to acquire a software support facility for use by NAVELEX software production groups. About $60 million is required for construction, computer hardware, software development, and personnel to staff the SSF.

The two modernization activities described above have several significant differences. The main difference is that TSSS is aimed at the support of a single program whereas SSF will support a number of NAVELEX-sponsored programs. These two support systems would not be likely to foster similar methodologies or uniform applications of standards. Also, tools from one are not likely to be transferable to the other.

Two modernization activities (out of many in the Navy) for sim/stim support systems are described below.

Integrated Simulation System - ISS is a distributed sim/stim system under development. It can drive simultaneous testing in multiple mock-ups. It will run on two to ten PDP VAX 11/780's connected in a bus architecture, and it will support the sim/stim needs of its facility.

Land-Based Integration Test and Training System - LITTS is a program in the planning stage. The main objective is to replace an older evaluation facility. Another objective is to design LITTS to replace the many sim/stim systems used by TRIDENT for sonar, development of defensive weapon systems, maintenance training, etc. The push for LITTS comes from the [illegible] to reduce the cost of the redundant sim/stim systems used by TRIDENT.

Hierarchical Development Methodology - An advanced methodology for software specification, design, and implementation. HDM supports formal verification (proof of correctness). It is being developed for the Navy by SRI International. The techniques have been used successfully on several production projects.

Performance Oriented Design - POD is a modeling tool under development for the Navy by BGS, Inc. It provides a means to assess the expected performance of a system, given the design in some state of development.

The projects described above and many others sponsored through Navy R&D are very promising; however, they will have no impact if they are not broadly transferred into production software operations. Technology transfer is crucial to the accelerated improvement in Navy software engineering practice.

REFERENCES

REFERENCES

Bailey, J.W. and Basili, V.R.: "A Meta-Model for Software Development Resource Expenditures", Univ. of Maryland Technical Report TR-936, August 1980.

Baker, F.T.: "Structured Programming in a Production Programming Environment", IEEE Transactions on Software Engineering, June 1975.

Basili V.R. and Turner, A.J.: "Iterative Enhancement: A Practical Technique for Software Development", IEEE Transactions on Software Engineering, Vol. 1, No. 4, December 1975, pp. 390-396.

Bisby, R., and Hollingworth, D.: "A Distributable, Display-Device-Independent Vector Graphics System for Command and Control", Information Sciences Institute Report ISI/RR-80-87, July, 1980.

Boehm, B.W.: "Software Engineering", IEEE Transactions on Computers, Vol. C-25, No. 12, December 1976.

BGS Systems, Inc.: Performance Oriented Design Preliminary Design Document, May 1978.

Brooks, F.P.: The Mythical Man-month, Addison-Wesley Publishing Company, 1975.

Dolotta, T.A. and Mashey, J.R.: "An Introduction to the Programmer's Workbench", Proceedings Second Int. Conference on Software Engineering, October, 1976.

Fagan, M.E.: "Design and Code Inspections to Reduce Errors in Program Development", IBM Systems Journal, Vol.15, No. 3, 1975.

Ferrentino, A.B. and Mills, H.D.: "State Machines and Their Semantics in Software Engineering", Proc. 1977 COMPSAC Cong. on Computer Software Applications, Chicago, pp. 242-251.

Fox, J.M.: Software and It's Development, Prentice-Hall, March, 1982.

Guttag, J.V.: "Abstract Data Types and the Development of Data Structures," Communications of the ACM. Vol. 20, No. 6, June 1977.

Halstead, M.: Elements of Software Science, Elsevier Computer Science Library, 1977.

Heninger, K.L., Kallander, J.W., Shore, J.E., and Parnas,D.L.. "Software Requirements for the A-7E Aircraft", NRL Memorandum Report 3876, November 1978.

Howden, W.E.: "Functional Program Testing", IEEE Transactions on Software Engineering, SE-6, No. 2, September 1976.

Jackson, M.: "The Jackson Design Methodology", IEEE Tutorial on Software Design Techniques, IEEE Catalog No. 76CH1145-2C, 1977.

Liskov, B.H., and Zilles, S.N.: "Programming with Abstract Data Types," Proceedings of ACM SIGPLAN Symposium on Very High Level Languages, SIGPLAN Notices (ACM) 9, April 1974.

McCabe, T.J.: "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. 2, No. 4, December 1976, pp. 308-320.

Miller, E.F. and Melton, R.A.: "Automated Generation of Testcase Datasets", Proceedings, 1975 International Conference on Reliable Software.

Mills, H.D.: "Software Development", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976.

Orzech, L.S.: "PSL/PSA: A Computer-aided Tool for Specification and Analysis of High Level Designs" IBM Software Eng. Exchange Vol. 2, No. 1 October 1979.

Parnas, D.L.: "Designing Software for Ease of Extension and Contraction", IEEE Transactions on Software Engineering, Vol. SE-3, No.2, March 1979.

Perriens, M.P.: "Using QBE to Process Software Requirements and Specification Data" IBM Software Eng. Exchange, Vol. 2, No. 1, October 1979.

Putnam, L.: "A General Empirical Solution to the Macro Software Sizing and Estimating Problem", IEEE Transactions on Software Engineering, SE-4, No. 4, 1978.

Ross D.T. and Schoman, K.E., Jr. "Structured Analysis for Requirements Definition" IEEE Transactions Software Engineering, January 1977.

Stevens, W.P., Myers, G.L., Constantine, L.L.: "Structured Design", IBM Systems Journal, 1974.

Teichroew, D. and Hershey, E.A., III, "PSL/PSA: A Computer-aided Technique for Structured Documentation and Analysis of Information Processing Systems." IEEE Trans. Software Engineering, January 1977.

Weinberg, G.M.: The Psychology of Computer Programming, New York, Van Nostrand Reinhold, 1971.

Wilson, M.L.: "A Semantics-based Requirements and Design Method", IBM Technical Report 03-072, General Products Division, Santa Teresa Laboratory, September 1979.

GLOSSARY

The purpose of the Glossary is to explain the terms, phrases and acronyms as they are used within this document. The definitions are provided to promote understanding; they are not offered as a standard for terminology within the Navy or industry.

ness and consistency of a product when compared against its predecessor; a review must have a broad range of participants and must have specific objectives or questions which are to be addressed. (Heninger [78])

[illegible] - functions expected of the [illegible] the operating system at run time; an Ada environment [illegible] [illegible] language.

[illegible] - Structured Analysis and Design Technique, a trademark [illegible] SofTech, Inc.

[illegible] parameters and input data sequences [illegible] simulation.

[illegible] software.

[illegible] and organizational relations, as shown by the [illegible] generate schedules by organizational entities.

SEATECS - the Software Engineering Automation for Tactical Embedded Computer Systems project which has the objective of designing and implementing a SEE to improve FCDSSA software production.

SEE - an integrated, computer-based support system and data base. A SEE is designed to improve the effectiveness of the software development process by creating more reliable software; it uses a well-designed software engineering process with proven methodologies and flexible, well-designed tools.

semantic analysis - evaluation of semantic information to determine the completeness and consistency of the information.

semantic language processor - software designed to check syntax of a specific formal language and to record it for

release - a self-contained copy of a system (or system component) which can be installed in the field (normally a project has several releases over its life cycle); a master relation that identifies all data objects belonging to (controlled by) a release.

reliability - the ability of a functional unit to operate without failure for a stated period of time under stated conditions.

reliability model - a support tool for correctness analysis; this tool uses the error history of the system over its life cycle to estimate reliability based on attributes such as size and complexity.

report generator - a generalized tool for creating reports; parameters are used to define the content of the report and the data source, and then the report generator software automatically creates the report.

requirement - a binding condition which states a mandatory characteristic of an abstract or physical object; requirements may have different form: a specific description, a constraint, an evaluation criteria for judging quality, or implied by context.

requirements analysis - analysis of the feasibility of requirements while at the specification and design level, and prior to development of the system.

resource - a means to accomplish a task; a resource, in the management sense, refers to the money, time, people or facilities available to accomplish a task.

resource estimation model - a management tool which uses the measurements from past projects, along with a characterization of the system to be developed, to generate resource estimates for planning.

reusable component - a software component performing a specific function and deliberately designed for reuse, with options or variations in function to be selected via parameters; reusable components would be of known high quality, with function and parameters well-documented, and freely available on system libraries.

review - a methodology used to gauge the internal complete-

quality assurance or QA - the process of ensuring that a process, system or software meets pre-defined standards.

query - a specific request by an interactive user for data, instructions or status; the request will elicit a response from the system.

queueing theory - the mathematical field for analyzing systems characterized by queues (items to be 'processed') and servers (processors).

record - a collection of related data items treated as a unit.

recovery - the activities associated with regaining a prior correct state after an error has occurred.

redirection - the ability to define (as a non-standard sequence) what program will receive the output of a program which has ended either normally or abnormally; pipelining deals with standard sequences, redirection with non-standard sequences.

re-engineer - the process of upgrading the design and implementation of existing systems.

reflow - moving words from one line of text to another to put as many words as possible between a line's boundaries; reflow is used after continuous typing to create blocks of text, or after changing line boundaries, or to justify ragged-right text.

rehostable - capable of being moved to a different host computer and becoming operative quickly with the installation of the host-specific kernel.

relational data base - a data base in which data objects are organized as a collection of rectangular tables, each table being a relation; users produce new relations by combining old ones (by the use of a non-procedural data manipulation) and do not need to know the structure or format of the data base, especially the physical structure.

regression testing - a testing technique which only tests the function which has changed.

executed separately to produce a specific result.

processor - the component of a computer which is capable of sequentially and non-sequentially interpreting and performing executable machine instructions.

product - the end result of an activity within the software engineering process; for example, a specification is the product of requirements analysis

productivity - a measure of output per unit of resource applied.

program - a sequence of instructions designed to make a computer perform a specific function.

program overlay - part of a program which, when loaded into a computer's memory, will overlay or replace all or part of a previously loaded segment or data of a program.

programmer's work space - the resources (notably data space) controlled by an individual for performing their work assignment.

program skeleton - a partly-created program which, with simple extensions, can be used to create a new program.

project - an organizational entity with an assigned development responsibility; a macro relation that identifies all data objects belonging to (controlled by) a project.

proof technique - rigorous techniques for proving a program correct or incorrect; proof techniques can be either formal or informal; formal proofs use systematic logical analysis and proof tables to derive correctness of prime programs, while informal techniques use a series of rigorous correctness questions.

prototype - the creation and study of an abstract model which will allow evaluation of requirements and design approaches.

PSL/PSA - Problem Statement Language/Problem Statement Analyzer.

QBE - Query-by-Example (tm.) - IBM's relational data base query language.

PERT - Program Evaluation and Review Technique; a specific tool for determining critical path and resource measures in a complex project environment.

physical structure - how data is physically organized and structured on a data storage device.

pipelining - the ability to define the normal output of one program as the normal input to another program, both programs being considered a standard sequence; redirection deals with non-standard sequences.

POD - acronym for Performance Oriented Design, a modeling tool under development for the Navy.

portability - a software component is portable when it can be moved to another host without any change to the component; it is rehostable if a different host-specific kernel is required.

portable - software which requires only the host-specific kernel to run on a different host computer.

power - the ability of a programming language to handle sophisticated problems simply and directly.

PPS - Program Performance Specification - describes the performance specification for a given software item on a given computer system.

preprocessor - software which does a preliminary computation or organization of source statements before it is compiled or assembled; some source statements contain preprocessor statements which, when executed by the preprocessor, alters the source statements.

pretty printer - software which formats listings to make them easier to read and understand.

private data - data in an individual's work space which is controlled by the individual.

procedure - a set of unique instructions for performing a specific function.

process - any unique sequence of instructions which can be

the output of a compiler or assembler; object modules can be executed directly on the computer but are normally connected with other object modules by a linkage editor to produce load modules.

off-line - pertaining to peripheral devices (such as magnetic tapes or disks) which are not directly connected to, or accessible by, the processing unit of a computer system.

on-line - pertaining to peripheral devices which are directly connected to, or accessible to, the processing unit of a computer system.

operating system - system software that enables a data processing system to supervise its own operations, and optimize the use of all hardware and software resources by automatically invoking system programs, application programs, language processors, and data to ensure continuous processing of a series of jobs.

operators and operands - technique for measuring the complexity of a low level design (Halstead [77]).

over-constraining requirements - requirements which constrict the latitude needed by designers to consider alternative approaches, thus, possibly, excluding an excellent design approach.

patch - executable code (either absolute machine code, or translated machine code from source statements) used for making an immediate fix to a software fault.

path test analyzer - a tool used to examine possible paths through a segment of code and to generate test cases for these paths.

PDL - Program Design Language; a methodology for formally recording a program design; PDL is sufficiently clear to support direct coding but flexible enough to leave some questions unanswered while proceeding with design.

PDR - Preliminary Design Review - a formal review of the early high-level design of a system.

performance - the capacity of a system to achieve a desired result; a productivity measure for a system.

model - a technique by which an abstraction of system is created so that it can be analyzed, and different sets of variables evaluated; models are usually mathematical in nature but can be semantic.

module - module is used very specifically in this document to mean: a component of the system-level design which is defined through the technique of information-hiding, and decomposes into other modules or into programs and data structures.

MTASS - Machine Transferable AN/UYK Support Software; a set of software used for support and kept current by the Navy.

mutation - a micro relation that refers to version variations of a system.

NADC - Naval Air Development Center.

NCCS - Naval Command and Control System.

NAVAIR - Naval Air Systems Command.

NAVELEX - Naval Electronics System Command.

NAVSEA - Naval Sea Systems Command.

nesting - the process of embedding structures, data, or subroutines, within others at a different hierarchic level.

non-baseline data - temporary information which is useful to the task at hand but is not worth keeping permanently; as an example, debug information from a test run is essential to the testing process but is not needed once the test succeeds.

non-procedural techniques - techniques for problem solving which do not require an established sequence of steps or instructions.

NOSC - Naval Ocean Systems Center.

NUSC - Naval Underwater Systems Center.

object module - executable machine instructions which are

media, that can be sensed by machines designed for that purpose.

macro relations - those relations between items in a data base which encompass a large number of data base objects; the nine macro relations are: baseline products, non-baseline products, measurements, archive, system support library, project, release, version, and increment; also see micro relations.

mailbox - a message technique used to communicate between individuals or groups.

maintenance - see adaptation.

management - the function concerned with all activity involved in the development and adaptation of software.

mapping - a correlation between two sets of data; e.g., the format of a system specification would be based on the semantic categories and relations chosen to define the system requirements.

MAPSE - Minimum Ada Programming Support Environment.

measurement data - data objects of a quantative nature representing some characteristic of the software development process; measurement data is used by management to assist in the planning and control of a project.

methodology - a repeatable human procedure or set of procedures that is used to translate data, which is input to a life cycle activity, into data which is the output of the activity; a well-conceived methodology provides for the separation of the creative, intellectual aspects of an activity from the more clerical, mechanical aspects.

micro relations - those relations between items in a data base which have significance in the context of a small number of data base objects; the micro relations discussed are: traceability, derivation, ancestry, version (mutation of a system or release), and association; also see macro relations.

milestone - a checkpoint marking the completion of a significant, clearly identifiable activity or task.

interpreter - software which executes the instructions of an interpretive language one at a time; APL is an interpretive language.

ISS - Integrated Simulation System

job - a unit of work for a computer; a job normally includes all of the system commands, programs, linkages, and files needed to perform a complete unit of work.

job control language - a problem-oriented language used to identify and specify all operations to be performed for a job.

LAMPS - acronym for Light Airborne Multipurpose System.

language analyzer - a tool for analyzing a specification written in a formal language.

life cycle - the set of activities, and the relations among these activities, involved in the development, operation, and continuing adaptation of a software system, beginning with its conception and continuing to its removal from use.

linker or linkage editor - system software which creates strings or sequences of executable code by resolving cross references and connecting separate units of object code.

LITTS - Land-Based Integration Test and Training System.

load module - programs in the form appropriate for execution by a computer; load modules are the output of a linkage editor after it processes the appropriate object modules.

loader - system software which reads data and executable code into a computer's memory.

log - a sequential file which contains time-ordered transactions against a data set.

machine code or machine language - see object code.

machine-readable - properly structured data, on a suitable

implementation - the software development activity concerned with the creation, unit test, and integration of code fulfilling a design.

increment - a management unit of a project associated with incremental development (one or several increments may equal a release); a macro relation that identifies all data objects belonging to (controlled by) an increment.

incremental development - the concept of defining and building software in small pieces, with each succeeding one resulting in a viable operational system with more function.

independent - a quality of software by which it is free from the influence of specific features offered on a computer system (hardware) or an operating system (software).

index - a list of the contents of a data base and containing references to locate a specific item.

information-hiding - a technique involving the isolation of information within modules; module boundaries are defined by the particular information to be isolated, such as design decisions and data definitions. Information-hiding decisions are made on the basis of expected changes to information thus localizing the effect of future changes to one module.

inspection - a technique for evaluating the correctness of a design, specification, code segment, test plan, etc.; inspections involve a small group of specific people following a well-defined procedure (Fagan [75]).

instrumented analysis - a methodology which captures data dynamically (during execution) and uses it to identify and locate errors.

integration - the software development activity which combines parts of a system to create a viable, operable system.

interactive - a mode of operation whereby each entry by a [illegible] response [illegible] computer [illegible] user.

interface - a shared boundary between two components of a system over which data passes.

graceful failure - a failure which only incapacitates a part of a system rather than the entire system.

graphics - a pictorial means of presenting information; usually on paper but can also be on an interactive device which displays graphics.

hardware simulator - software which duplicates the operational characteristics of a hardware device.

HDM - Hierarchical Development Methodology; a methodology for specifying, designing, and implementing software; it also supports formal verification.

help mechanism - an interactive feature which allows the user to be prompted on how to perform an operation, or how to solve a problem.

heuristic - literally, helping to discover or learn; a heuristic process modifies itself based on past performance, while making progress toward an acceptable result; progress is based on a series of approximations toward that result. Heuristics is currently synonomous with artificial intelligence (AI) and is used in knowledge-based expert systems.

hierarchic data base - a data base in which data objects are organized in a specific hierarchy relative to one another; users must know and understand the hierarchy to manipulate data with ease.

high-level language - a programming language whose instructions do not bear a one-to-one relationship with the generated machine instructions; a high-level language may either be a problem-oriented language like FORTRAN or a universal language such as COBOL.

host or host computer - a general-purpose computer system with extensive support software and capacity; software should be developed on a large scale, well-supported host rather than a target computer unless the target computer is large scale and well-supported; once the software is developed it would be loaded into the target computer for operational use.

FASP - acronym for Facility for Automated Software Production, a large support system, which uses commercial computers and was developed by Naval Air Development Center (NADC).

failure - the degradation of a system when an error occurs; see error.

fault - an incorrect implementation of the specification; see error.

FCDSSA - Fleet Combat Direction Software Support Activity in San Diego (FCDSSASD) and Dam Neck (FCDSSADN).

feasibility analysis - an evaluation to determine whether there exists a design which satisfies the requirements within system constraints.

file - a collection of related records treated as a unit.

firmware - software residing in read-only memory.

foreground - an operating system usually has two modes of operation: background and foreground; foreground is biased toward servicing short, interactive work requests on a high priority basis, and background is biased toward larger work requests (jobs) which are queued and serviced on a scheduled or FIFO (first-in-first-out) basis as system resources come available.

formalism - the mathematical or logical structure of an argument as distinguished from its content.

formatting tool - a tool for transforming software, or a document, into a specific format.

FORTRAN - acronym for FORmula TRANslation, one of the first high-level languages which was commercially available; FORTRAN was designed for writing programs to solve problems which could be stated mathematically.

function - a rule of correspondence between two sets such that, for each element in the first set, there is one and only one element in the second set to which it corresponds. In relation to software, a function is the rule of correspondence between two sets such that in the first set represents input and the second set represents output.

entry point - the point within a program to which control is passed by another program; each executable program should have one entry point and one exit point.

environment simulator - a testing tool which simulates an operational environment.

error - a discrepancy between the correct value or condition, and the actual value or condition. An error is a manifestation of a fault during execution and may cause a failure; a fault is an incorrect implementation of the specification; and a failure is the degradation of a system when an error occurs.

error-day - the average time (in days) a potential error-causing fault goes undetected.

error handling - see exception handling.

estimation model - software which uses a data base of measurements on performance of past projects, along with a description of the new project, to generate estimates for the development activities of a new system.

exception handling - the special processing triggered by an abnormal (usually erroneous) condition.

executable design - a design in a formal language which can be interpretively evaluated to determine the design's completeness, performance and correctness.

extendability - the ability to add, easily and with minimum rework, new functions to an existing system.

external interface - a shared boundary between a component of a system and a component considered to be outside the system.

external reference - a program reference to a symbol outside of the program; external symbols and external references are resolved either statically by a linkage editor or dynamically by a loader.

external symbol - a symbol defined in an owning program as available to other programs outside the owning program.

the specification, provides a description for implementation, and is feasible within the appropriate constraints (time, schedule, cost, etc.)

development - all the activities needed to specify, design, implement, test, document, and manage the creation or alteration of software.

diagnostics - warnings or error messages pertaining to the isolation and detection of a potential problem, malfunction or error.

directed graph - consists of a set of nodes and directed links connecting nodes, and has an implied direction; a PERT chart is a directed graph; an acyclic directed graph is a directed graph in which no sequence of links leaving a node ever returns to it.

distributed data base - a data base whose components can be geographically separated yet, conceptually and operationally, are considered part of the same data base.

down-line load - the process of moving data or programs from a host computer to a target computer over a channel-to-channel link.

driver - code that simulates the activity of a superior software component while testing a subordinate component.

dump - the mass transfer of data from one medium (normally internal) to another medium (normally external) either as a safeguard against errors or as an aid to debugging.

dynamic analysis - analysis techniques which examine an item by running it in a known environment; the correctness analysis activity uses dynamic analysis through a variety of testing techniques with known inputs and known environment.

ECP - engineering change proposal; a requested change to hardware or software.

ECS - Embedded Computer System; a Navy system containing an embedded computer and requiring software support.

EIA - Electronic Industries Association.

variable, and a different program accesses the variable and uses it.

data dictionary - a central collection of definitions describing the names and attributes of data elements in a data base.

data flow analysis - a design methodology which uses the movement of data in a system to decompose functions or modules.

data object - abstraction for the attributes and values of data.

data reduction - the processing performed on data to transform it from one form to another.

data structure - a technique for organizing data into a more convenient form for manipulation without regard for its actual physical structure, e.g., a record of 30 data elements is more convenient to process than 30 separate data elements.

DBMS - see data base management system.

debug - an informal term used for the process of detecting and eliminating faults in programs.

decomposition - the process of breaking an element into smaller pieces, each of which can be further decomposed independently of the others.

default - one of a set of alternative values or options that is assumed when none has been specified.

derivation - a micro relation which provides the ability to relate data objects because one is explicitly derived from another.

design - the creative activity of describing a system through the techniques of abstraction and decomposition. Abstraction is used to suppress all but those details needed to understand the design at a specific level. Decomposition involves dividing the system into pieces (modules, programs and data structures) which can be related in some hierarchical fashion. The resultant design: solves the problem posed by requirements, is consistent with

crash - an informal term for the abnormal cessation of processing by a computer system.

creativity aid - any tool or technique which helps to develop an idea.

critical path - the shortest path in a precedence network; the path that represents the least amount of a critical resource in a development project.

cursor - a position marker on a video display.

cyclomatic complexity - a measure for complexity which uses graphs to study control flow. (McCabe [76])

C3 - command, control and communication.

DASD - contraction for direct access storage device, magnetic storage devices usually based on disk or drum technology and allowing direct access to a unit of data.

data base - the set of data required by an application, function or system; the data can be redundant or non-redundant depending upon the design of the data base; the data can be organized hierarchically (like an inverted tree), or relationally (showing how data is related to other data), or as a network.

data base management system or DBMS - software which will store, control access, retrieve, manipulate, update and purge information from a data base shared by many functions or applications in an organization. The value of a DBMS is its simplicity to the user, separating the logical attributes of the data from the physical attributes. A DBMS allows the user to concentrate on what needs to be done, while the DBMS handles all I/O operations and data manipulations. A DBMS generally includes: a data management function for storing and controlling access to data; a data dictionary for describing the data; a generalized query language for retrieving and massaging selective data; and a report generator for printing massaged data. The separation of logical and physical attributes permits programs which view the data logically to remain unchanged when the physical structure of the data is changed.

data binding - a relationship whereby a program modifies a

requirements have been met by the subsequent specification, i.e., that no inconsistencies or omissions exist in the specification which should be expressed in a formal language.

complexity - the sense of a whole having many intricate, interconnecting parts.

component - a part or element of a whole; software components are invariably complex parts within complex subsystems within complex systems.

concatenation - the process of connecting two or more strings in a specific order, e.g., CA concatenated with BD yields CABD.

configuration - the relative arrangement of components in a system.

configuration management or CM - a management activity which controls the changes made to a frozen (or "baseline") component or system, and thus protects it from unauthorized alteration.

control flow analysis - a methodology for analyzing the structure of control decision points within software.

control structure - a small set of logic elements used to create proper programs; a proper program has one entry point, one exit point and well-defined input and output; DO, DO-WHILE, DO-UNTIL, IF-THEN-ELSE, and CASE are commonly considered the basic control structures.

conversion - the process of changing from one data processing system to another, usually with a different instruction set architecture.

correctness - the concept that a software product satisfies its functional and interface specifications; correctness is an objective to be balanced against other, sometimes contradictory, objectives but should be considered a necessity in the development of all software.

correctness analysis - a comprehensive set of methodologies to ensure that software correctly implements its specification and that faults are detected and removed soon after they are introduced.

character set - an agreed-upon set of elements used to represent, control, or organize data; ASCII and EBCDIC are examples of standardized character sets.

CMS-2 - Compiler Monitor System-2; a Navy standard program which translates the source statements of the problem-oriented or high-level CMS-2 language into object code.

CDR - Critical Design Review; a review of the detailed design of an element of software held when detailed design is complete.

code - loosely, a computer program; the term may represent: (1) the original instructions (source statements); or (2) the resultant instructions created by compiling and assembling the source statements into executable object code; or (3) the act of creating (coding) a computer program--this last use of code includes, loosely, the activities of detail design, code and unit test.

code translator - software which converts programs from one programming language to another programming language.

command language or CL - a source language consisting of operators which evoke functions to be executed.

commonality - the concept that similar function exists in every software system.

compatibility - the concept that different computers can run the same software without appreciable alteration of the program.

compilation - the transformation of a program, written in a problem-oriented language, into an equivalent programming language which is oriented to a specific machine; the original program may be compiled independently of the transformed program, which is usually processed by an assembler.

compilation units - separate units of a connected set of software programs which may be compiled independently.

compiler - software used to transform source programs into another programming language closer to the machine.

completeness analysis - the technique of ensuring that all

batch job - a job which normally processes sequences of similar records; batch jobs normally run in background but can be run in foreground if needed; batch jobs are usually queued to be processed in FIFO (first-in-first-out) order.

batch processing - the processing of accumulated, similar units of information, e.g., a set of payroll records.

bit - contraction for Binary digiT; a unit of information that has one of two values, either 0 or 1.

black-box - a technique which uses only external, observable attributes: the input is known, the resulting output can be observed, and the internal workings are not considered.

block diagram - a diagram of a system in which principal parts are represented by standardized geometric shapes to show the basic functions and interrelationships.

breadboard - a technique used to prepare an early working model or prototype; engineers once used yellow wire and blank circuit boards (or breadboards) to design circuits.

breakpoint - a place in a program where its execution may be externally examined, modified, and execution resumed or stopped.

buffer - a storage area used to compensate for the difference in speed between two functions or I/O devices during the movement of data.

byte - a sequence of adjacent binary digits (bits) operated on as a unit, and usually shorter than a computer word; a byte usually contains eight, and sometimes six, bits.

capacity - a measure of processing rate; the amount of data that can be contained by a data storage device.

change request tracker - a tool designed to simplify the control of change requests as they are entered, tracked and resolved.

channel - in information theory, that part of a communication system which connects the message source with the message sink.

and data formats.

assembly - the process of transforming source statements to object code through the use of assembler software.

assertion - a formal statement about the function of a code segment; assertions can be used for formal proofs of correctness or for dynamic verification during execution.

assertion analyzer - a verification tool which allows embedding of assertions in code; dynamic analyzers verify the assertions against actual software execution; static analyzers verify for syntactic or semantic correctness.

association - a micro relation used to relate data objects explicitly where the relationship is not obvious, such as a performance model with a specific system design.

audit - a methodical examination and review for a stated purpose.

automatic structurer - a tool which transforms unstructured code into structured code.

background - the low-priority mode provided by an operating system; see foreground for a more complete definition of foreground and background.

background job - a job which is run by the operating system in a low priority (background) mode when system resources are not being completely used by the high priority foreground mode, and the resources are available for use by the background mode.

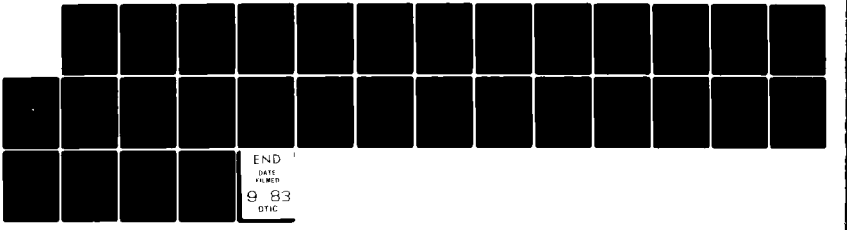
background job stream - a sequence of queued jobs which are started and processed automatically by the operating system.

backup - a copy of an original file, data base or system, which is kept as a safeguard against the loss of the original.

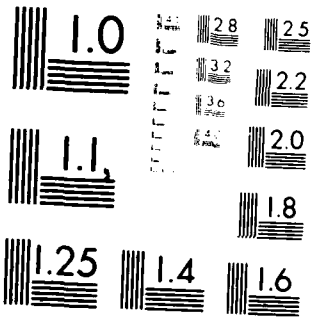
baseline - the body of data about a system which defines it at a specific point in time, such as Release 1.1 as of 12/18/82; the body of data against which changes can occur under management control.

AD-A131 941 A SOFTWARE ENGINEERING ENVIRONMENT FOR THE NAVY(U) 3/3
NAVAL MATERIAL COMMAND WASHINGTON DC 31 MAR 82

UNCLASSIFIED F/G 9/2 NL



END
DATE FILMED
9 83
DTIC



abstract data - data with non-specific attributes; see abstract data type.

abstract data type - a concept which assigns to data a set of high-level, abstract attributes (allowable operations and tests), until it can be given more specific attributes by the process of stepwise refinement.

abstract interface - a concept which reveals only the information needed to use an interface and nothing else; a black-box definition.

abstract machine - any combination of hardware and software which provides different characteristics than either alone; a concept that a machine is the total of its hardware and its software.

abstraction - the process of focusing on selected qualities of a complex subject by suppressing all but those details needed to understand the design at a selected level.

access controls - a mechanism for determining who can access what data, and what operations can be performed on that data.

acquisition management model or ACM - a life cycle model used to plan and manage all activities for the development of software systems.

activity - a category of related tasks from the software engineering process; the term activity is synonymous with the phrase "life cycle activity."

Ada - a modern high-order language for programming which will become the standard language for DOD's mission critical applications.

adaptation - the process of making changes to existing software either to add new function or to correct an embedded fault; adaptation is comparable to the more common term "maintenance" or the Navy term "in-service engineering;" it is not comparable to the term "life cycle support" which includes all activities in the life cycle of software.

alphanumeric display - a device designed to display alphanumeric (A to Z, 0 to 9) characters plus a limited set of

further use.

semantic model - a description of the elements of a user's environment and its specific requirements, using a rigorously-defined syntax for categorizing terms and establishing relations among the categorized terms. Requirements, once they are precisely defined in a formal syntax and recorded, can be analyzed with a variety of semantic information tools.

set - a fundamental concept in mathematics of a well-defined list, collection or class of objects. The objects in sets are called the elements or members of the set and can be anything: people, rivers, dates, numbers, etc.

SHARE/7 - a multiprocessor time-sharing system hosted on AN/UYK-7 computers.

simulation - a technique for dynamically evaluating a design before implementation is started; simulation requires a model and a simulator.

simulator - a tool for simulation.

software - Broadly defined: the information needed to perform a desired function on a computer (the "hardware"). Software includes the sequence of instructions to be executed on the computer, the related data for the sequence of instructions, and the instructions for the operating system. (Note that the application data to be transformed by the function is not included in the definition nor is documentation such as user and operator instructions.) Narrowly defined: that sequence of executable computer instructions and related data needed to control the execution of a computer to perform a desired function. The executable instructions must be in a form suitable for loading into a computer; these instructions then control the sequence of instructions needed to perform the function. Related data can be either real or abstract. Real data has actual content and can be manipulated, e.g., AVERAGE = 15.467 or DATE = 1 MAY 1982. Abstract data defines what the data looks like, i.e., its form and structure. (Notes: The term "related data" does not include application data. The narrow definition of software is an attempt to be unambiguous and implies that software, in its purest sense, cannot be seen nor touched; it also excludes instructions for the user and operator's instructions.)

software development - a phase of the software life cycle including requirements analysis, specification, design, implementation, correctness analysis, and management.

software component - a general term for software which performs a specific function and has a unique identifier; component can refer to a system, a subsystem, a routine, or a segment of code.

software engineering - a discipline based on sound engineering principles and proven methodologies, for creating software that is correct, reliable and economical; the application of repeatable methodologies and proven disciplines to the design and implementation of software.

software engineering process - the overall set of activities, applied throughout the life cycle of software, which produces the required information needed at the interfaces of each life cycle activity.

source statements - the original sequence of instructions for a unit of software (usually written in a high-level language) before it is transformed into object modules.

specification - the product of requirements analysis which transforms the less rigorous system requirements into a rigorous description of the external view of a system.

SSF - Software Support Facility, a proposed NAVELEX software production facility.

standards checker - a static analysis tool which examines code and documentation for violations of standards.

state machine - a function defined in terms of state transitions which transforms a state with input into a new state with output. More formally: a function (set) whose members are ordered pairs of ordered pairs: m = [(x, y), (u,v)] , where each member of m has the interpretation ((<state>, <input>), (<next state>, <output>)).

static analysis - analysis techniques which involve a detailed manual examination of an item; the correctness analysis activity used three static analysis techniques, here given in order of increasing rigor: reviews, inspections and proofs of correctness.

stepwise refinement - the process of decomposing a unit into successive levels of greater and greater detail; each decomposition is checked for correctness before continuing to the next lower level.

structured design - a methodology for creating a design: data flow diagrams are analyzed and the structure is derived by applying design rules and guidelines.

structured programming - a programming technique which views a program as a functional composition of a finite small set of one-in one-out control structures.

stub - a component which simulates the functioning of a subordinate component during testing of the superior component.

support system - a set of tools (preferably integrated) which are designed to aid software engineers in manipulating information on a data base.

symbolic execution - a technique for proving the correctness of certain classes of programs by executing them symbolically to prove certain assertions.

synchronous - two or more processes that depend upon the occurrence of a specific event; having a regular or constant time interval.

syntax - the rules concerning the pattern or structure of the word order in a formal language.

syntax checker - a tool for examining statements and validating their correctness against the rules of the formal language.

system generation or SYSGEN - the process of selecting optional software components and creating a particular operating system configuration for a specific installation.

system test - an independent verification that the system meets specifications.

target computer - the computer system which executes ECS software.

table-driven code - code whose execution is controlled by entries in a table or set of tables.

test - execution of a system (or component) with known input to a known environment with expected results.

test data generator - a tool which automatically generates test data based on the design (or the code) and additional instructions such as "test every path through 'A'."

test case - data designed to test a single path of a software component.

test harness - a tool which provides a framework for testing small units of code; it is an interactive tool for defining the procedure interface, preparing test data, running the instrumented test, and displaying the test results.

text editor - a tool for creating and manipulating prose in machine-readable form.

throughput - a measure of the speed with which a computer system will process a unit of work from a mix of jobs from input through output.

throwaway code - the creation of code to prove a design but which may have to be discarded because of what is learned from the creation of the first version; analogous to bread-boarding in hardware.

time tag - the time-of-day and date information appended to data.

tool - a tool, in this document, is a computer-based mechanism that either stimulates the creative, intellectual process, or else automates a clerical process; it is a mechanism used to perform all or part of a methodology.

tool atoms - building blocks which alone perform no useful task but, in combination with other tool atoms, can be used to create useful tools.

tool kit approach - an approach to software engineering wherein a set of tools (possibly integrated) is provided to the software engineer but no constraints are implied on the choice of methodologies or tools to be used.

top-down implementation - an incremental implementation approach stressing the creation of an operable, but skeletal, system early in the life cycle; since the system always works, stubs can be replaced with their completed component, and the system retested with all effort focused on integrating only one component; top-down implementation allows one to deal with a single component's problems more gracefully, and eliminates the need for drivers but requires stubs.

traceability - a micro relation which provides the ability to follow a requirement as it impacts specification, design, code and other baseline products.

tree - an acyclic directed graph starting from a single node.

trouble report or TR - a mechanism for describing a defect and tracking it.

TSSS - Trident Software Suport System; an ECS software development system using standard commercial software and hardware and supporting code generation, unit test and documentation.

Type A specification - states the technical and mission requirements for a system as an entity, allocates requirements to functional areas, and defines the interfaces between or among the functional areas.

Type B specification - requirements for the design or engineering development of a product (system component) during the development period.

unit test - the testing of the lowest unit of software independently of other programs which interract with it.

uses hierarchy - a hierarchy of modules wherein the relations of higher level modules to lower level modules is one of dependency, wherein the related lower level modules must correctly implement their respective specifications in order for the higher level module to function properly.

validation - an activity that assures that each product reflects the requirements and specifications applicable to it.

verification - a process that assumes that a product implements its design.

version - a variation of a release; a macro relation which identifies varying current implementations of a system which may be required to support multiple (but slightly different) physical installations; a micro relation which identifies the mutation of a data object.

Warnier-Orr diagram - a graphic technique for displaying control flow in software.

WBS - work breakdown structure - a technique of decomposing tasks into sub-tasks with resource and personal assignment of responsibilities.

white-box - a testing technique which uses knowledge of the internal design of a program, i.e., the input is known, the internal workings of the program is known, and the resulting output can be observed.

6.2 - Program 6.2 for Exploratory Development within the Navy's RDT&E (Research, Development, Test and Evolution) accounts.

INDEX

stract data type - I: 34; II: 52; G: 1.
abstract data structure - I: 30-31; II: 6.
abstraction - I: 2, 29-30, 32; G: 1.
acquisition management model - I: 12-14; G: 1.
Ada - Intro: 1, 3; I: 37, 39-40; II: 53, 60-61; G: 1.
adaptation - I: 2, 22, 31; II: 1, 9, 36; G: 1.
ancestry - II: 24; G: 2.
application paradigm - II: 21, 57-58; G: 2.
application skeleton - Intro: 4; I: 56-57; G: 2.
architecture - II: 27, 38, 60-61; A: 3, 11; G: 2.
assertion analyzer - I: 47-48; II: 54; G: 3.
audit - I: 52; G: 3.
background - Intro: 1-3, 7; II: 34, 39, 41; G: 3.
black-box - I: 45-47; G: 4.
commonality - I: 56; II: 3; A: 11; G: 5.
completeness analysis - I: 27-28; A: 4; G: 5.
configuration management - I: 9, 16, 49, 52-53, 55; II: 5, 9, 20, 33; G: 6.
control structure - I: 37-38; II: 29, 45; G: 6.
correctness analysis - Intro: 5; I: 4, 6-7, 12-13, 15, 25, 27, 30, 41-50, 52; II: 14, 54, 56; A: 4; G: 6.
critical path - I: 53-54; G: 7.
cyclomatic complexity - I: 35; G: 7.
data base management system - I: 23, 37, 56; II: 20, 40, 60; G: 7.
data binding - I: 35; G: 7.
data dictionary - I: 20-21, 23; II: 43, 52; G: 8.

data structure - I: 23, 29-31, 34; II: 6; G: 8.

decomposition - I: 29, 30-34, 44; II: 52; G: 8.

derivation - II: 26; G: 8.

dynamic analysis - I: 44, 49; G: 9.

error - I: 6, 12, 28-29, 33, 36, 39-41, 49-50; II: 4, 7, 14, 18, 20, 30, 32, 40-41, 45, 59, 62; G: 10.

error-day - I: 41, 44; G: 10.

estimation model - I: 24; II: 55; G: 10.

executable design - I: 9, 46; G: 10.

feasibility analysis - I: 21-22; G: 11.

HDM - A: 12; G: 12.

hierarchic - II: 13, 22-23; G: 12.

high-level language - I: 24, 38; G: 12.

information-hiding - I: 30-31, 34; II: 52; A: 12; G: 13.

inspection - I: 44-45, 47, 49, 52; II:33, 54; G: 13.

integration - Intro: 1; I: 4, 7-8, 12-13, 37-38, 49; II: 48; G: 13.

interface - Intro: 2, 4, 5-6; I: 26, 30, 33, 48-50, 56; II: 2, 6, 27-39, 42, 48, 57-58, 60-62; G: 13.

macro relations - II: 4, 22-23; G: 15.

mailbox - II: 11; G: 15.

mapping - I: 20, 26; G: 15.

micro relations - II: 4, 22-26, 40; G: 15.

model - I: 2, 4, 6-9, 12-13, 15-16; II: 11, 25, 55; A: 11; G: 16.

non-procedural - Intro: 4; II: 29-30; G: 16.

patch - II: 7; G: 17.

PDL - I: 30, 32-35; II: 6, 45, 52, 56, 60; G: 17.

performance - I: 2, 13, 19, 21-22, 24, 28, 35-37, 40, 46-47, 49; II: 2, 22, 25-27, 34, 37, 54, 58-59, 62; G: 17.

pipelining - II: 30, 40, 48-49, 56, 62; G: 18.

POD - II: 60; A:12; G: 18.

portability - II: 27, 49.

productivity - Intro: 1-4; I: 4, 7, 38, 50, 55-56; II: 57; A: 1, 4, 6; G: 19.

program skeleton - II: 21, 57; G: 19.

proof - I: 27, 44-45, 47-49; II: 9, 54; A: 12; G: 19.

prototype - I: 9, 24, 46; II: 12, 50; A: 12; G: 19.

quality assurance - I: 9, 52, 55; II: 9; A: 4; G: 20.

redirection - II: 16, 18, 40; G: 20.

relational - I: 22-23, 25; II: 22-24; G: 20.

reliability - Intro: 1-3, I: 1, 6, 38, 47, 49, 55; II: 16, 18, 54, 58; A: 4; G: 21.

requirements analysis - Intro: 4-5, I: 4, 6, 8, 12-13, 18-19, 22-28; A: 5; G: 21.

reusable component - Intro: 4; II: 57; A: 4-5; G: 21.

review - I: 13, 15, 44-45, 47, 52, 55; II: 11; A: 2; G: 21.

SADT - I: 20, 33; G: 22.

SCR - A:12; G: 22.

semantic - I: 19-27; II: 5, 12, 16-17, 24, 26, 50, 56, 62; G: 22.

static analysis - I: 44, 48-49; II: 7, 13; G: 24.

stepwise refinement - I: 32, 34; II: 52; G: 25.

structured design - I: 32; G: 25.

structured programming - I: 37, 39; II: 45; G: 25.

symbolic execution - I: 47-48; II: 54; G: 25.

system generation - I: 37-38, 40; II: 4, 20; G: 25.

system test - I: 4, 46, 50; II: 8, 37, 54; G: 25.

top-down - I: 38; II: 53; G: 27.

traceability - I: 16, 20, 22, 26, 29; II: 6, 23-24, 26-27; G: 27.

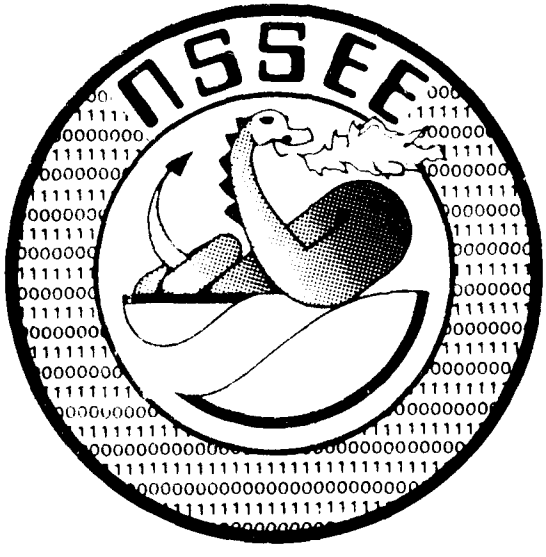
uses hierarchy - I: 32, 34; II: 52; G: 27.

verification - I: 6, 8; A: 12; G: 28.

version - I: 4, 6, 9, 31, 36; II: 7, 18-20, 22-24, 37; G: 28.

white-box - I: 45-46; G: 28.

EVOLUTION PLAN FOR A NAVY STANDARD SOFTWARE ENGINEERING ENVIRONMENT



SOFTWARE ENGINEERING ENVIRONMENT

WORKING GROUP

March 31, 1982

The logo for the Navy Standard Software Engineering Environment (NSSEE) shows a sea serpent riding the crest of the software technology wave and breathing structure into the software development process (represented by the well-ordered field of ones and zeros). NSSEE should be pronounced "Nessie" in honor of a lake-dwelling resident of the British Isles. History does not record any interaction between Nessie and another well-known British citizen, Ada. The modern Ada and NSSEE will be well-acquainted, however.

PREFACE

This is one of a set of documents resulting from the work of the Naval Material Command's Software Engineering Environment Working Group (SEEWG). The set includes an "Executive Summary," "Framework for a Navy Standard Software Engineering Environment," and "Evolution Plan for a Navy Standard Software Engineering Environment."

The Framework document provides a basis on which to plan and develop a standard methodology-driven software engineering environment for the Navy. The Evolution Plan describes a strategy for transition to a standard software engineering environment and also the evolution of the environment itself.

These documents represent the culmination of the first phase of an effort to develop standard tools and procedures to support development of software for Navy embedded computer systems. It is intended ultimately to implement a standard software engineering environment encompassing the whole software life cycle and supporting effective use of the Ada programming language. In the interim, these documents will provide a basis for co-ordinating decisions with respect to improvements to existing software support systems.

FOREWORD

This document contains the outline pertaining to the evolution to, and evolution of, a Navy Standard Software Engineering Environment (NSSEE). An expansion of this outline into a finished document will be completed by October, 1982. The work was done for the NAVMAT Software Engineering Environment Working Group (SEEWG).

The principal investigator in this effort was Robert N. Charette, Naval Underwater Systems Center, Newport, Rhode Island. Others that reviewed this document and provided valuable inputs were: LCDR K. Paige (PMS 408), T. P. Conrad (NUSC), T. E. Dawson (NUSC), G. Bain (NUSC), R. House (NUSC), T. Phillips (NOSC), H. Stuebing (NADC), S. Peele (FCDSSA), and C. Russ (FCDSSA).

I. INTRODUCTION

Construction of a Software Engineering Environment (SEE) for the Navy will be a very difficult task. It will require several orders of magnitude more effort than that which will be expended to implement Ada. Therefore, a plan is needed to evolve user and application projects/software support systems into a SEE, and, while the SEE is evolving, to allow these groups to make use of software engineering concepts as they are developed.

This document is one of a series produced by the NAVMAT Software Engineering Environment Working Group (SEEWG). The first document, "Framework for a Navy Standard Software Engineering Environment," presents the goal of a standard software engineering environment in terms of the methodologies and tools required to support the Navy's software life cycle model. The second document, a "Strategy Document," discusses the programmatic issues of how to establish and manage a SEE. This programmatic information is contained in a memorandum from the SEEWG to MAT 08Y and is not available for general review. The "Evolution Document" discusses the technical issues and policies, standards, and guidelines necessary to transition to a NSSEE, as well as the subsequent evolution of a NSSEE.

II. EVOLUTION TO A NAVY STANDARD SOFTWARE ENGINEERING ENVIRONMENT (NSSEE)

A. Technical Approach

1. Description of the Evolution Plan

a. Constraints

i) The NSSEE is to be based on the MAPSE. The MAPSE effort, producing a minimal set of tools to support Ada, will yield the Navy's most modern standard software engineering tool set. It is the most appropriate starting point for the evolution of a NSSEE.

ii) Existing application projects and support systems cannot be ignored and must be able to benefit from the concepts and products resulting from each step in the development of the NSSEE.

b. Plan Outline

i) The life cycle model in the document "Framework for a Navy Standard Software Engineering Environment" defines several life cycle activities. The interface between activities will be defined by the Navy in an interface specification document controlled by the Navy.

ii) The Navy will build at least one integrated tool set that will operate through all activities and interfaces. These tools and interfaces will comprise the NSSEE. A minimal tool set necessary to produce

baselined data at each life cycle activity interface will be provided. The completeness of the interface definition, coupled with the ability of the provided tool set to satisfy these interfaces, should eliminate the necessity of using any tools other than those provided within the NSSEE. However, the NSSEE will incorporate supplementary tools for each life cycle activity that may, depending upon the project, be beneficially applied. For example, within the "specification" activity, Figure 1, a language analyzer tool is provided as well as one (or possibly several) modeling tools and consistency/completeness analysis tools. Thus, within the NSSEE, there will exist flexibility for tailoring tool selection to satisfy application dependent needs and still meet interface standards.

2. Implementation Issues of a NSSEE

   a. An incremental approach will be used to develop the NSSEE with the Navy MAPSE as the first increment. Initially, work will be directed at the interface point that is known best, i.e., the code/test interface. The initial definition of interface requirements is defined in "Framework for a Navy Standard Software Engineering Environment" under the heading Baselined Products. Figure 2 illustrates the incremental approach, showing the growth of both tool set and interface data requirements until the the ideal life cycle is supported by the NSSEE.

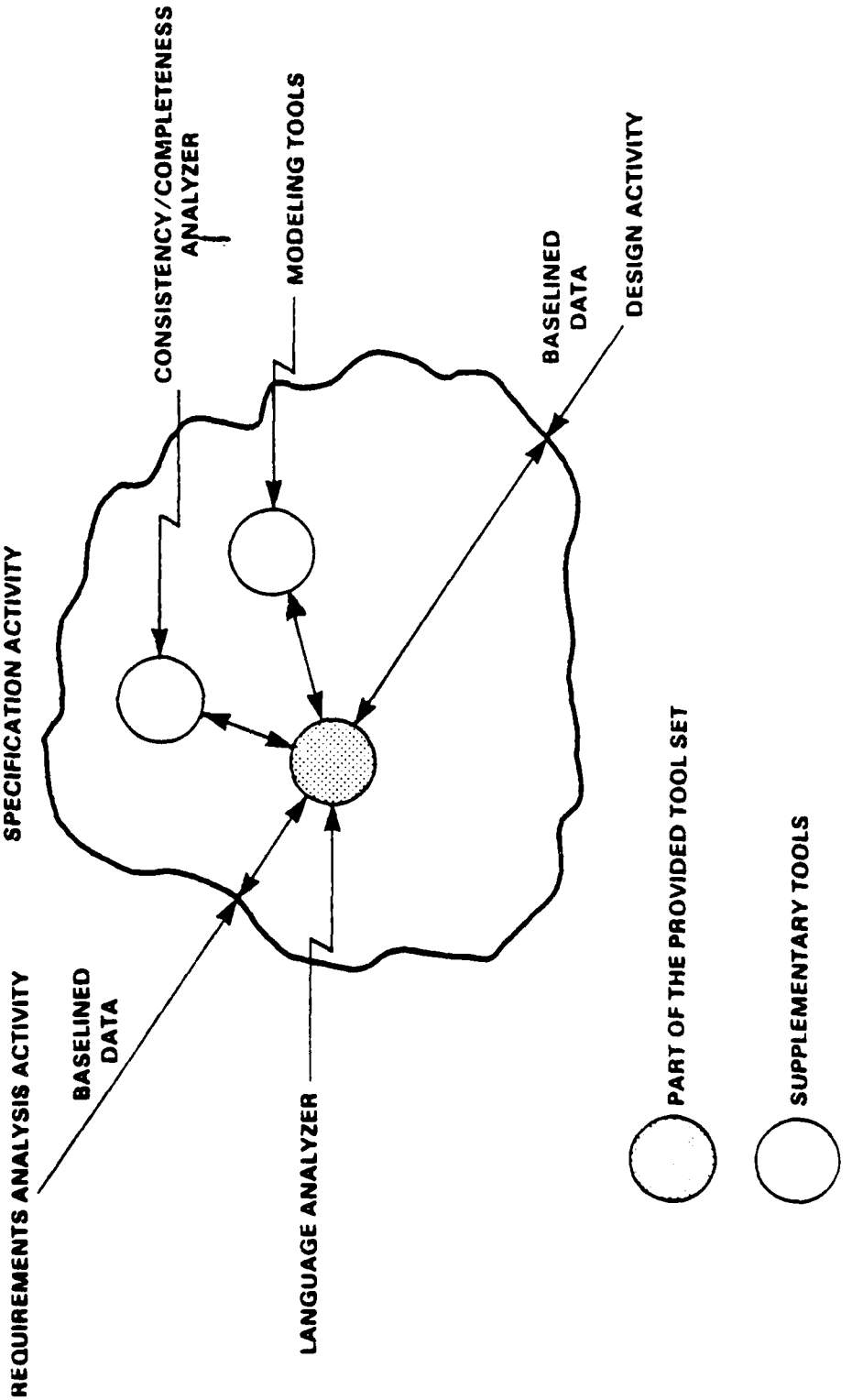


FIGURE 1

TOOLS/DATA RELATIONSHIPS WITHIN A LIFE CYCLE ACTIVITY

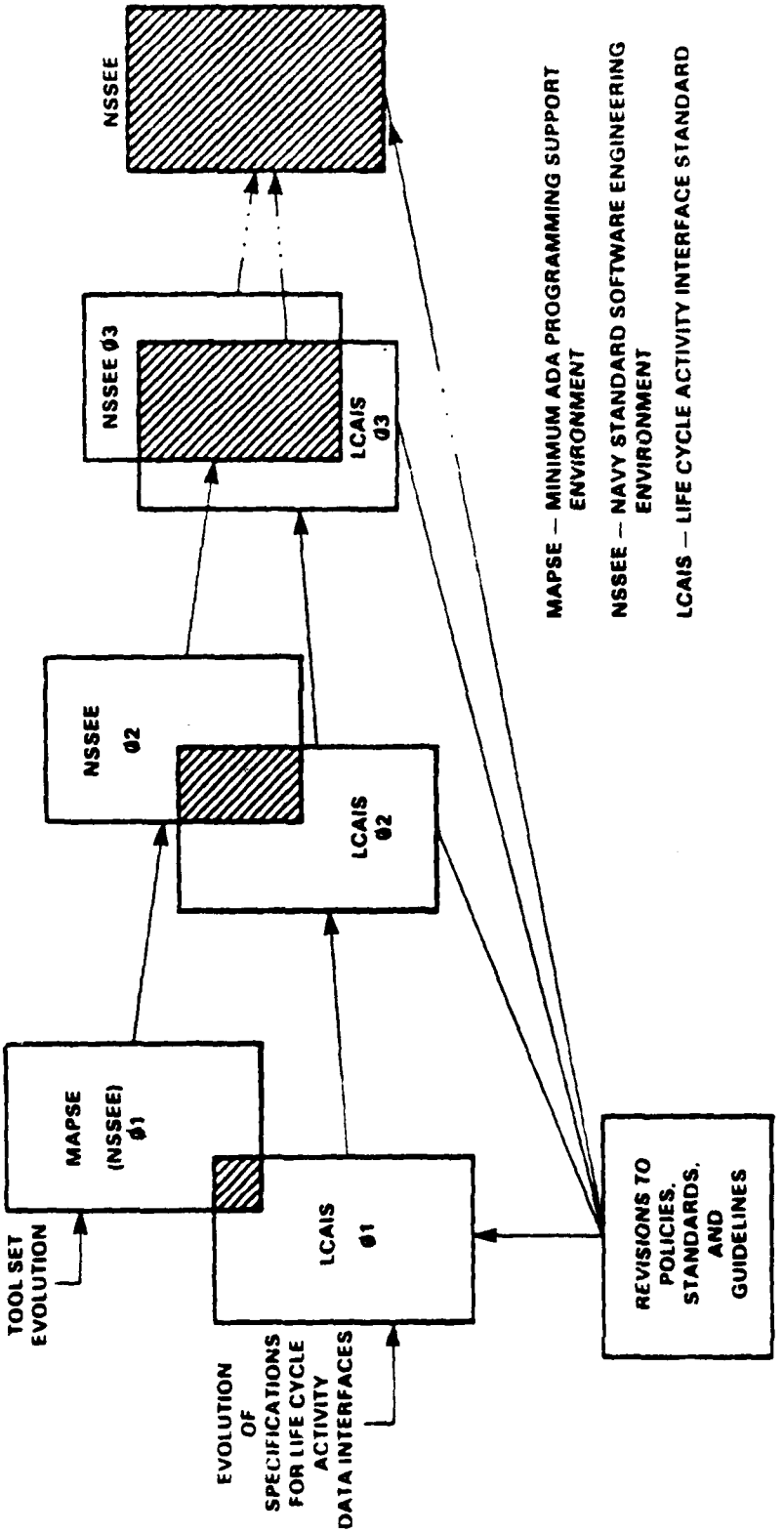


FIGURE 2

EVOLUTION TO A NSSEE

b. Implementation of the "complete" NSSEE (as defined in the Framework document) is beyond the current state-of-the-art, thus, research efforts will be required. Research areas need to be identified and prioritized. Areas of research called for in the Framework document include:

- methodology (which one(s) to use within and across life cycle activities).
- interfaces (syntax, semantics, numbers of, etc.).
- tools (types required, build, buy, etc.).
- quality assurance.
- software reusability.

c. Impacts of Incremental Approach

i) The incremental approach provides the benefits of a standard software engineering environment in the near term while providing for the phased introduction of the results of R&D efforts as they become available. It also facilitates the graceful migration of current application projects and support systems to the NSSEE by providing for the gradual adoption of standard products and procedures as they are produced.

ii) As new interfaces and tools are added, there may be unavoidable perturbations on previous work completed on the NSSEE. Orderly expansion and careful research efforts should help minimize

these impacts. The exact nature of this expansion and under whose supervision are unresolved issues.

d. The issue of training/education is very important for managers, developers, and users of the NSSEE. The understanding of how to competently use the NSSEE is critical to any benefits derived from it. The areas of concern are training new and retraining "old" personnel (both government and contractors) and making the user interface into the NSSEE as simple as possible. To alleviate these concerns the following will be required:

- Complete documentation that is tutorial in nature.
- A human engineered, user-friendly interface for the NSSEE.
- NAVMAT supported "Tiger Teams" to resolve on-site installation and "learning curve" problems as specified in the Strategy Document.

B. Policies, Standards, and Guidelines to support this approach.

1. There is an immediate need for modification of, or additions to, existing Policies, Standards, and Guidelines in order to have a framework under which a NSSEE can be built.

a. New Policies, Standards, and Guidelines recommended are:

1) The PDA should designate PMS 408 as the DA, and FCDSSASD as the

LCSA (MAT 08Y is the PDA) for the NSSEE.

ii) The interface specification should be a basis for a TADSTAND. However, the specific characteristics of the interface must be reflected in a mil-standard. The Framework document should be used as the baseline reference for the generation of a Navy standard software engineering environment TADSTAND.

iii) MAT 08Y shall promulgate "Framework for a Navy Standard Software Engineering Environment" as a companion to the Embedded Computer Resources Master Plan.

iv) MAT 08Y should use the Framework document as the context for all decisions/plans concerning the NSSEE, including:

- Coordinating all current and future software engineering environment related efforts.

- Research and development decisions.

Once the NSSEE has been specified and implemented, no other software engineering environment efforts will be undertaken by the Navy, except those for the NSSEE.

v) Guidelines for coordination with Joint Logistics Commanders' and the AJPO work are required. These are discussed more fully in the Strategy document.

vi) Standard licensing agreements

for commercial tools that may become part of the NSSEE should be drafted.

vii) The need for a waiver from OMB Circular 109 allowing for Navy prescription of software engineering tools and procedures on contracted software acquisitions should be investigated.

b. Existing Policies, Standards, and Guidelines that may require modifications follow. Specific changes and the justifications for each remain to be documented.

i) Acquisition:

- SECNAV 5000.1A
- SECNAV 5200.32
- TADSTAND 2, 3, 9, A, B, C, D

ii) Documentation:

- SECNAVINST 3560.1
- TADSTAND E

iii) Security/Privacy:

- OPNAVINST 5239.1
- OPNAVINST 5510.1F
- SECNAVINST 5211.5B
- NAVMATINST 5510.17

iv) Quality Assurance:

- NAVINST 4855.1A

v) Transfer of Responsibility for Navy Tactical System Software:

- NAVMATINST 5200.27A

vi) Update the Master Plan for Embedded Computer Resources

2. Once the first interface specification is approved as a standard, the initial element of a NSSEE exists. Thus, MAT 08Y must determine whether it will allow waivers from this standard and also whether it will provide funding to facilitate compliance with the standard. During the transition period, prior to implementation of the full NSSEE:

   a. New software support systems (i.e., NSSEE oriented and approved projects) will be required to meet this standard.

   b. Existing software support systems, running existing application projects (e.g., Share/7 supporting MK117 FCS) should be considered for a waiver from this standard on a case by case basis after a cost/benefit analysis is completed.

   c. New application projects will be required to use support systems that meet the standard. If it is desired to use old support systems (e.g., avionic system "AIM-IX" running on FASP), then these support systems must be upgraded. As a consequence of this approach, these new application projects will be able to migrate more easily to support systems that accommodate the latest iteration of the interface standard.

   d. Determining compliance with the standard interface specification implies an "accreditation" mechanism

of some type. Accreditation procedures (i.e., how the semantic and syntactic consistency of the interfaces is maintained), and who is responsible, need resolution.

3. In time, additional interfaces and standard tools will be incorporated to "flesh-out" the skeleton NSSEE. MAT 08Y, based on recommendations of the NSSEE program and management organization as described in the Strategy document, will then need to determine which support systems and/or application projects are required to upgrade to the standard tools and interfaces.

   a. New application projects will be required to use the standard tools and interfaces.

   b. Contractors will be required to deliver products that meet the interface specs, but not necessarily to use the standard tools.

   c. Existing support systems will be upgraded on a case-by-case basis.

4. After a "complete" NSSEE (as defined in the Framework documents) is available, all new Navy projects will be required to use it.

   a. No funding for upgrades to old support systems will be allowed. This ensures one NSSEE.

   b. [illegible]

c. Contractors will be required to deliver products that meet the interface specifications, but not necessarily be required to use the NSSEE. This will allow contractors to be innovative in tool selection and use (e.g., to gain or maintain a "competitive edge") which in the long term should be beneficial to the Navy. The NSSEE will be GFE to any contractor in response to direction from a Navy sponsor.

5. Milestones

A schedule of milestones for the development of the NSSEE and applicable policy and standards revisions must be defined.

III. EVOLUTION OF THE NSSEE

A. Continued evolution of the NSSEE is required to ensure the transfer of benefits as the technology (hardware, software engineering systems, etc.) evolves, experience is gained, and/or requirements change as shown in Figure 3.

1. The sources of new technology will be from inside and outside the Navy community, as described in the Strategy document. The "Not Invented Here" and "maintain the status quo" syndromes must be avoided.

2. A "software change control board" will be incorporated to determine how, when, and what new technology should be added to the NSSEE.

3. Care must be exercised to minimize the probability of a need to modify the interface standard. All Navy projects using the NSSEE may have to be upgraded if the interface standard is modified. In that event, procedures must be developed to facilitate such upgrades.

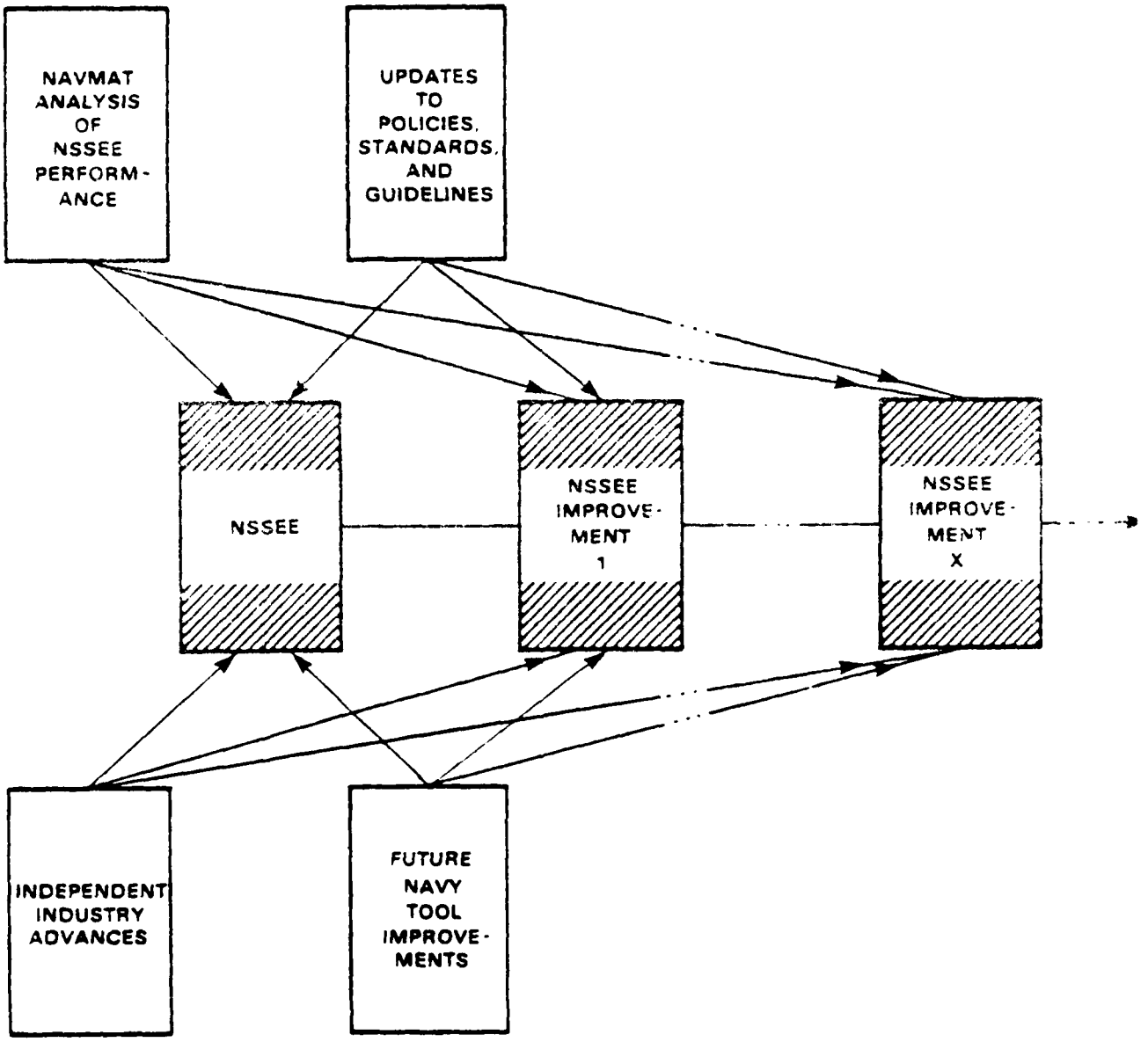


FIGURE 3

EVOLUTION OF THE NSSEE

END

DATE
FILMED

9-83

DTIC